Portable, Modular Expression of Locality

David Stoutamire

TR-97-056

December 20, 1997

Abstract

It is difficult to achieve high performance while programming in the large. In particular, maintaining locality hinders portability and modularity. Existing methodologies are not sufficient: explicit communication and coding for locality require the programmer to violate encapsulation and compositionality of software modules, while automated compiler analysis remains unreliable.

This thesis presents a performance model that makes thread and object locality explicit. *Zones* form a runtime hierarchy that reflects the intended clustering of threads and objects, which are dynamically mapped onto hardware units such as processor clusters, pages, or cache lines. This conceptual indirection allows programmers to reason in the abstract about locality without committing to the hardware of a specific memory system. Zones complement conventional coding for locality and may be added to existing code to improve performance without affecting correctness.

The integration of zones into the Sather language is described, including an implementation of memory management customized to parameters of the memory system.

Dedication

For my thoughtful and loving Patti Jean. With you I have found a home.

This has been a tale of too many cities. My work is all about closing distance, and it won't be complete until our beloved son Bruce returns from Jamaica.

Special appreciation for Steve Omohundro, creator of Sather and an great teacher; Jerry Feldman, my most patient advisor; and compadre Ben Gomes. I also extend warm thanks to everyone on the Sather team with whom I had the pleasure of working.

My father began my journey with geodesic domes, agar, sphagnum and orchid sex. My mother's support has never wavered. It is a great blessing to be born of educators.

May we all live with wisdom, simplicity, and grace.

I would think until I found
Something I can never find,
Something lying on the ground,
In the bottom of my mind.

- Anne Morrow Lindbergh

Contents

INTRODUCTION 8

The Problem 8
Portable performance 9
Modular performance 10

Towards a Solution 11
Cooperating for locality 11
Overview 12

MEMORY SYSTEMS AND LOCALITY 14

Hardware 14
Going the distance 15
Uniprocessor issues 16
Multiprocessor issues 21

Software 26
Why misses happen 26
Locality maintenance 28

PERFORMANCE MODELS 32

Prior Models 33
Implicit models 33
Explicit models 37
Annotative models 39

Case Studies 40
Portable performance with registers 41
Parallel Memory Hierarchy model 42
Sather 1.1 distributed extension 47

ZONES 50

Pure Zones 51
The pure zone model 52
Zones in Sather 55
Comparison with other models 57

An Implementation 59
Organization 59
Design for locality 64
Performance results 68

EXTENDING ZONES 81

Hardware Zones 81
The hardware tree in Sather 83
Placement examples 86
Caching 87

Example: Distributed Vector 88
Creation 89
Vector addition 90
Random access 91
Exploiting iteration locality 92

CONCLUSION 94

Summary 94
Perspective on locality 94
Zones 95

Future Directions 96
Migration and scheduling 96
On-line performance feedback 97
Libraries 97

APPENDIX: THE SATHER LANGUAGE 99

Sather **99**
Language overview 99
Implementation overview 104

Threaded Extension **112**
par and fork statements 113
parloop statement 114

Synchronization Extension **115**
lock statement 115
Attach statement 118
sync statement 121
Memory consistency 122
SYS class 123

Distributed Extension **123**
The '@' operator 124
Location expressions 125
with-near statement 125

REFERENCES 126

Introduction

This chapter introduces the problem of building software systems that are portable, provide good performance, and yet can still be constructed without prohibitive effort. An overview of the work is provided. The following chapters will explain in more detail the shortcomings of the current ways such systems are built and how a new performance model can help, while later chapters motivate, justify and explain this new model.

THE PROBLEM

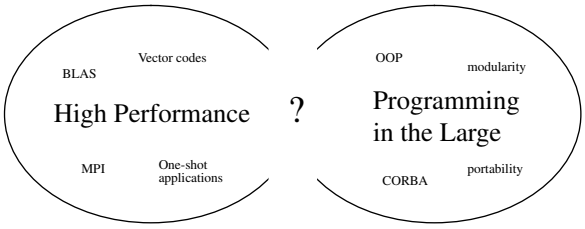
It is difficult to attain high performance while programming in the large.

High performance drives much of the effort being poured into computer development. When computer time is expensive, it makes economic sense to put a lot of expensive programmer effort into speeding up a program because it will pay off many times over, each time the program is run. High performance is the only reason for programmers to worry about obtaining parallelism, and often can justify the writing of specialized, single application code designed to run on a specific machine.

Programming in the large is required when code is too large or complicated to be reasoned about in its entirety by a single individual.

- Many codes require groups of programmers or groups of groups, and thus a way to split up the code into modules. This way, no single individual has to understand the entire system at once: modules can be reasoned about independently. Code that is split up in this way, with formally defined interfaces that make it possible to know that the system will work as a whole when the pieces are put together, is *modular.*
- Many codes live long enough or are important enough that they need to be maintained by someone other than the programmer who wrote them. In this case there is also more than one person involved - in essence, teams of programmers are working together separated in time. This kind of programming in the large over time introduces additional requirements. The ability to understand and maintain code may be more important than the convenience of writing it. Codes that live a long time need to be able to run on different hardware as it becomes available; they must be *portable*.

Some code requires high performance, and some code requires programming in the large. The problem of providing one or the other has been examined in detail, and methodologies exist for each. However, at this time it is not easy to do both at once, illustrated by the figure below. What is needed is a common methodology. The next sections examine problems that arise when trying to solve both problems at once.



Portable performance

High performance and portability are at odds. High-performance demands detailed hardware models, while portability requires abstractions that filter out hardware details. Both high performance and portability require extra insight and effort from programmers. As a result, the few high performance systems that are portable are also monstrously expensive in programmer resources.

	Application	
English		English
	Algorithms	
Fortran + *SLP*		Sather + *zones*
	Program	
LAPACK		Class library
	Libraries	
BLAS		Sather Runtime
	Inner loops	
ISA		ISA
	Hardware	

Figure 1: Abstraction layers of programming. Each layer has an interface: on the left are the interfaces as conceived in [6] (page 42); on the right are corresponding levels presented here.

The center column of figure 1 depicts various abstraction levels necessary for building software. At the bottom is hardware, which is too difficult to program directly; each step above represents a simplification of the one below, leaving out details. High performance

requires reasoning at the lower levels. Portability must avoid reasoning at the lower levels which change with the hardware. Design at multiple levels at once is very difficult and the mark of a seasoned systems developer.

Reasoning at multiple levels is eased by defining interfaces that decompose the software modules that implement upper and lower functionality. For example, programming languages isolate the details of instruction sets from upper levels, and LAPACK and BLAS [8] isolate application code from the details of the implementation of linear algebra operations. In this way, new lower levels can be plugged in without modifying upper levels. This approach works well for fixed applications in which a relevant abstraction of hardware can be created.

Unfortunately, there are many applications that require algorithmic changes to achieve high performance across a range of hardware [5]. This implies that for portable upper abstraction levels, the interfaces all the way up to algorithms must reflect those characteristics of hardware that could guide algorithm selection. For example, Alpern [6] suggests language level changes (space-limited procedures, page 45) to allow portable performance across very different memory systems. More generally, portable performance requires cooperation between all system components, including the operating system, application code, compiler, runtime and memory management, as well as hardware. This work attempts to enable this cooperation. It generalizes Alpern's work to object-oriented programming, and allows modeling a wider range of hardware systems, including networks of workstations.

Modular performance

High performance and modularity are at odds. Modularity makes it possible to decompose a program into smaller pieces, each of which can be reasoned about separately. Correctness of an entire program is then reduced to the correctness of each component in isolation and the correctness of their composition. For high performance, correctness means that in addition to producing the desired result, the program can do so without requiring too many hardware resources over space and time.

Unfortunately, performance now often depends not only on *what* is done - the algorithm - but also on *when* and *where* it takes place within the hardware. For example, a multiprocessor allows many tasks to be accomplished at once, but not if all the tasks try to run on the same processor at the same time. Recent computer designs make this worse with new resource constraints, such as cache memory, that weren't major performance bottlenecks a decade ago. Using these resources unwisely can result in catastophic slowdowns, such as losing all the benefits of parallelism.

Managing resources requires matching what the software needs - the resource requirements - with the resources available. When there is more than one software component, the best match of software to hardware cannot be done by reasoning about one isolated component at a time, because doing so ignores information about what resources the other components need. So just as correctness requires formally defined interfaces, performance

correctness requires interfaces that abstract the resource requirements of the components. Components with such an interface can then be composed without unduly losing performance due to poor use of hardware resources.

TOWARDS A SOLUTION

Reconciling performance, portability and modularity depends on resolving problems of locality. There are also other major problems that are not examined in this thesis; for example, interoperability between heterogenous hardware and software components. Locality is sufficiently difficult to warrant studying in isolation.

Cooperating for locality

The essential strategy proposed here for dealing with locality is to enable various system components to cooperate with each other. For example, the application writer may have an idea about the access patterns of the data structures being operated on in a program. Meanwhile, the operating system tries to exploit locality in the page replacement policy, and the compiler tries to improve locality in the form of optimizations such as loop fusion and blocking. However, there is rarely any communication between these components; each must work in isolation.

This isolation is no longer necessary. Figure 2 shows some of the ways that components can communicate with each other. Programmers can inserts pragmas into the code - hints to the compiler about how to do the best job - and this gives more information than is encoded in the program structure alone. Similarly, some operating systems accept hints about paging, so they don't have to guess entirely on the basis of past paging behavior. These extra sources of information allow components to cooperate. While useful, these hints are piecemeal, esoteric, system-specific and clumsy. What is needed is a unified approach to locality, in which communication between components isn't point-to-point and system specific. Locality within a component must be described in a way useful to other components and not tied to specific hardware.

This work presents a language-level programming model that makes thread and object locality explicit, based on first-class *zones*. Zones form a runtime hierarchy that reflects the intended relationships between threads and objects. The runtime may dynamically map zones onto hardware units such as processor clusters, pages, or cache lines to best cooperate with the memory system. Zones may also be used to extend memory semantics - for example, entires zones may be deallocated at once.

Locality is attained in uniprocessors by keeping similarly accessed data together. To avoid false sharing in multiprocessors, it is also important to keep unrelated data apart. This is why the word 'zone' was chosen. In a city, zones are areas in which construction, resi-

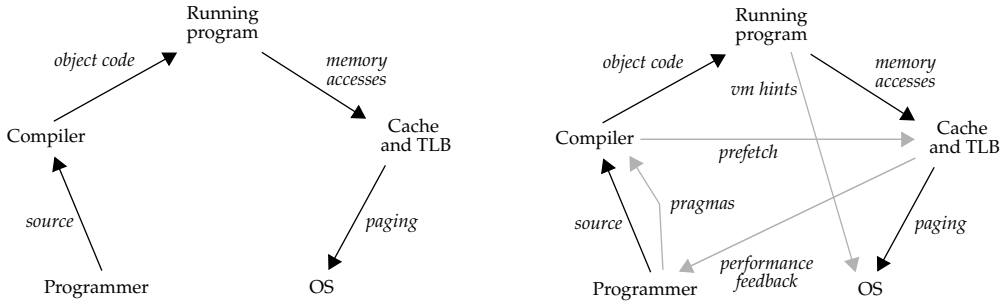


Figure 2: Flow of information about locality. Left, information in a traditional system flows away from the programmer, at each stage there being less to work with. On the right, a 'modern' system with extra, piecemeal information flowing between components.

dence, and establishments are regulated so that they do not interfere. Careful planning of the use of land allows a city to run smoothly. Similarly, a program must enable cooperation of its threads and objects to obtain high performance.

Overview

The two chapters which follow this introduction survey the existing state of the affairs in achieving high performance on modern memory systems.

- The following chapter, **Memory Systems and Locality (page 14)**, reviews the organization of memory system hardware. If current trends continue, there is the potential for severe performance problems in many applications due to poor use of bandwidth. The solution to this problem is to increase the effective use of on-chip memory. When on-chip memory is in the form of a cache, the way to make more effective use of it is through locality. Techniques are described to allow software to offer exploitable locality to the memory system without requiring programmers to adopt new ways of thinking about code.
- The traditional interface to memory allocation is not sufficiently expressive to enable portable locality. Many alternate programming methodologies have been suggested to make the programmer aware of the performance limitations of hardware. **Performance Models (page 32)** describes these and discusses their weaknesses. None of these models makes it possible to write code that is simultaneously portable, modular, high-performance and general-purpose.

The next two chapters introduce and explore a new performance model which attempts to adequately address locality.

- As an alternative to conventional memory management and thread primitives, **Zones (page 50)** introduces the zone performance model as a way to reason about software locality in the abstract. This makes it possible to construct modular software able to obtain reasonable performance on disparate memory systems. The way zones are expressed in the Sather high level language and the implementation of a locality-conscious memory manager is described, and the effect on application performance is examined.
- For the highest performance systems, it may be desirable to expose more details of hardware than the pure zone model allows. **Extending Zones (page 81)** proposes extensions to the pure model appropriate for systems with heavy penalties for poor locality, and implementation concerns when systems span more than one address space.

Conclusion (page 94) summarizes this work and suggests future avenues of research. Because this work builds heavily on Sather language and implementation at ICSI, an appendix **(Sather Language, page 99)** provides an overview of the language concepts and implementation.

Memory Systems and Locality

The previous chapter, **Introduction (page 8)**, explained problems that arise in programming in the large due to locality. This chapter begins with a discussion of memory system issues that limit application performance and an overview of hardware improvements to memory systems that do not assume software attention to locality. It is concluded that hardware approaches alone will not be sufficient for all applications, and that future microprocessor performance is likely to be limited by the memory system.

The chapter goes on to discuss ways that changes to software can improve performance when hardware solutions are not sufficient. Existing software techniques that improve the locality of data references are reviewed; these can reduce the required bandwidth as well as improve latency. Methods of software development that improve locality include automatic compiler optimizations, manual code transformation, and attention to memory management. The techniques presented in this chapter do not require programmers to use nontraditional languages or methodologies.

The following chapter, **Performance Models (page 32)**, discusses models which do require learning a new way to program, and for performance, force programmers to consider more of the hardware.

HARDWARE

This section reviews physical limitations that limit communication between components of computer systems. In particular, because fabrication technologies are advancing faster than packaging technologies, performance is increasingly limited by the need to communicate with memory off-chip, rather than by a lack of processing resources on-chip. This section reviews hardware approaches to dealing with this bottleneck.

Going the distance

Each generation of fabrication technology has been shrinking chip feature sizes by about 0.7x, making transistors cheaper and faster. Wires connect the transistors, so wire delay affects overall performance. Transmission delay was not a major concern for older technologies because it was a small fraction of clock cycle time. Now, as clock frequencies approach 1 GHz, interconnect delay is becoming a performance limiter and attracting the attention of circuit designers and fabrication process technologists.

The Alpha 21264 microprocessor, for example, has a large die (~300 mm^2) and a designed cycle time of 2 ns. As a result, its functional units must be arranged such that each communicates with only one or two others. An exception is the data cache, which must work with multiple integer units, the FPU, and the system interface. As a result, it takes two cycles to get an address to the cache and return the data. The problem is not in the cache array itself; access takes less than a full cycle, but there is no time to move the address or data any significant distance across the large die [46].

Recall that wire delay may be estimated by the RC (resistance-capacitance) product. Shrinking a chip from one generation's technology to the next leaves the delay of wires on a chip roughly unchanged, because the increase in line resistance from the reduced metal cross-sectional area is offset by a reduction in line length. However, overall die size and wire length are increasing rather than decreasing over time. Everything else held equal, for a fixed wire length the delay has been *doubling* with each generation.

Over the last few generations, technology improvements have kept the actual delay increase closer to 1.3x. The number of metal layers has been increasing by about 0.75 per generation. The wire aspect ratio has also been improved by about 0.22x per generation, reducing resistance. However, the number of metal layers will become impractical at around 10 in 2-4 generations, and continuing to increase the aspect ratio ratio will not bring much improvement because of increased capacitance and crosstalk [13].

Other technologies for reducing interconnect delay are on the horizon. Silicon dioxide is the standard dielectric in use today; a material with a higher dielectric constant would reduce capacitance, with the best alternative providing about a factor of two. The aluminum presently used for interconnects might be replaced by copper, which has about half the resistivity. Each of these improvements (more metal layers, improved wire aspect ratio, better dielectric, better conductor) may produce a constant improvement in interconnect, pushing back limitations for a small number of generations. Ultimately, however, these constant factors cannot compensate for the larger circuits being created. No matter what exotic technologies arise in the future, the speed of light will ultimately force the highest performance computations to respect distance, and locality will have to be respected in design.

Regardless of what is happening inside of chips, present technologies impose an additional severe penalty for communicating between one chip and another. Figure 3 compares microprocessor performance with corresponding pin bandwidth over time. Although this ratio also reflects changes in processor design, it is clear that bandwidth

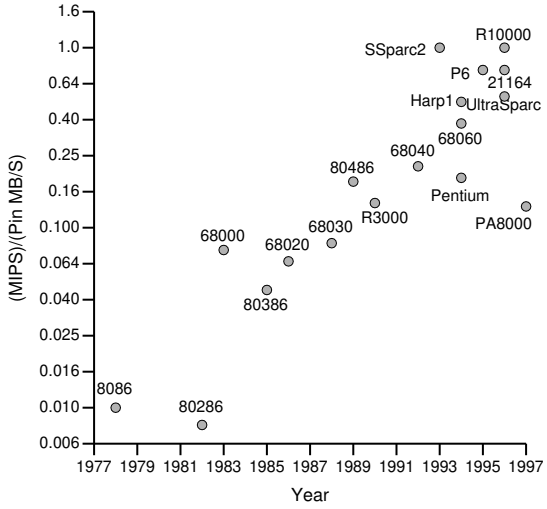


Figure 3: MIPS divided by with aggregate pin bandwidth (log scale, from [21])

from a chip to its environment is not increasing as fast as interconnects within the chips are. This gap is only likely to widen, because in addition to the electrical issues faced on-chip, packaging technologies also have to deal with heat dissipation and assembly constraints. With today's single chip processors, this delay shows up in the number of cycles that a processor needs to talk to memory and other processors. The effect of this increasing distance to memory has been most dramatic in the behavior of memory loads and stores. The next section looks at ways this changing behavior has affected processor designs.

Uniprocessor issues

Performance is lost when processing operations stall, wasting the opportunity to do work. This may occur because a memory operation was not issued sufficiently ahead of an operation that depends on it, in which case that operation is *latency-bound*. If the memory operation stalled because other memory operations congested a resource such as a shared bus, the dependent operation is *bandwidth-bound* [21]. The following sections describe hardware techniques for avoiding operations bound by latency or bandwidth.

Reducing latency

The latency penalty can be reduced by faster interconnects or increasing bus speed. Unfortunately, memory access times have been improving 5-10% slower per year than processor speeds. Another way to reduce latency is to move some accesses to memory that is faster because it is smaller and closer to the CPU. For example, register files are tiny, multi-ported on-chip memory small enough to be managed by the compiler.

Registers reside in a distinct address space from main memory. It is also possible to have an addressable fast memory (Cray-2), but this usually takes the form of a cache. Cache memory is managed by hardware which attempts to recognize systematic memory accesses. Future accesses are anticipated by keeping recently accessed data and, typically, speculatively prefetching words that are close to recent accesses. This is possible because the sequence of addresses is usually far from random; for most applications there are exploitable patterns. Three particular patterns of access often receive special hardware support:

> *Temporal locality* exists when the same location is referenced closely in time. This is exploited by attempting to keep recently accessed locations in the cache.
>
> S*patial locality* exists when addresses are referenced which are close to previous addresses. This is exploited by *prefetching*, loading nearby regions of memory into the cache instead of individual words, in the expectation that they will be needed soon.
>
> *Streaming* accesses are equal to a previous address plus a *stride*. This is common when accessing array data, and is exploited by predicting the stride and fetching locations ahead of the accesses, another kind of prefetching.

Processors typically interleave several address streams, one of which is an instruction stream. Instruction accesses are typically (but not always [85]) highly predictable; they universally warrant handling with a separate specialized cache. Instruction caches will not be considered further, and all mentions of cache should be understood here to mean either a data cache or a unified data and instruction cache.

Caches have proven so useful that most microprocessors employ multiple levels of cache, even within a chip, as well as other forms of cache such as virtual memory translation lookaside buffers (TLBs) and paging to memory from disk. Levels are denoted *L1*, *L2*, etc., as the size increases and bandwidth and latency deteriorate. Caches are organized as some number of *blocks*. Each block may hold a copy of a range of memory, and often 'block' is used to refer to the represented range. The *capacity*, or size, of a cache is the number of blocks times the number of bytes in the block.

Figure 4 demonstrates the effect of different cache levels on latency for three microprocessor systems. Average latency is measured as an aligned array of memory is strided through with each load dependent on the previous. As the size of the array increases, it ceases to fit in each level of cache and the latency rises. The block size can be inferred from the shape of the curve as the stride increases; there is exploitable locality when the stride is smaller than the block size, but once each access touches a new block, there is no longer any benefit to the cache at that level and there is an inflection point.

Caches can reduce latency only to the extent that the hardware prediction strategy is accurate, which depends on the program and input data. Cache utility can be optimized for a given workload by careful attention to trade-offs of size, associativity, and replacement policy. The transparency of caches comes at the cost of making it difficult for a system to treat cache memory as a resource to be carefully managed, as these hardware mechanisms come into play when accessing memory whether they are wanted or not [20]. (MIPS pro-

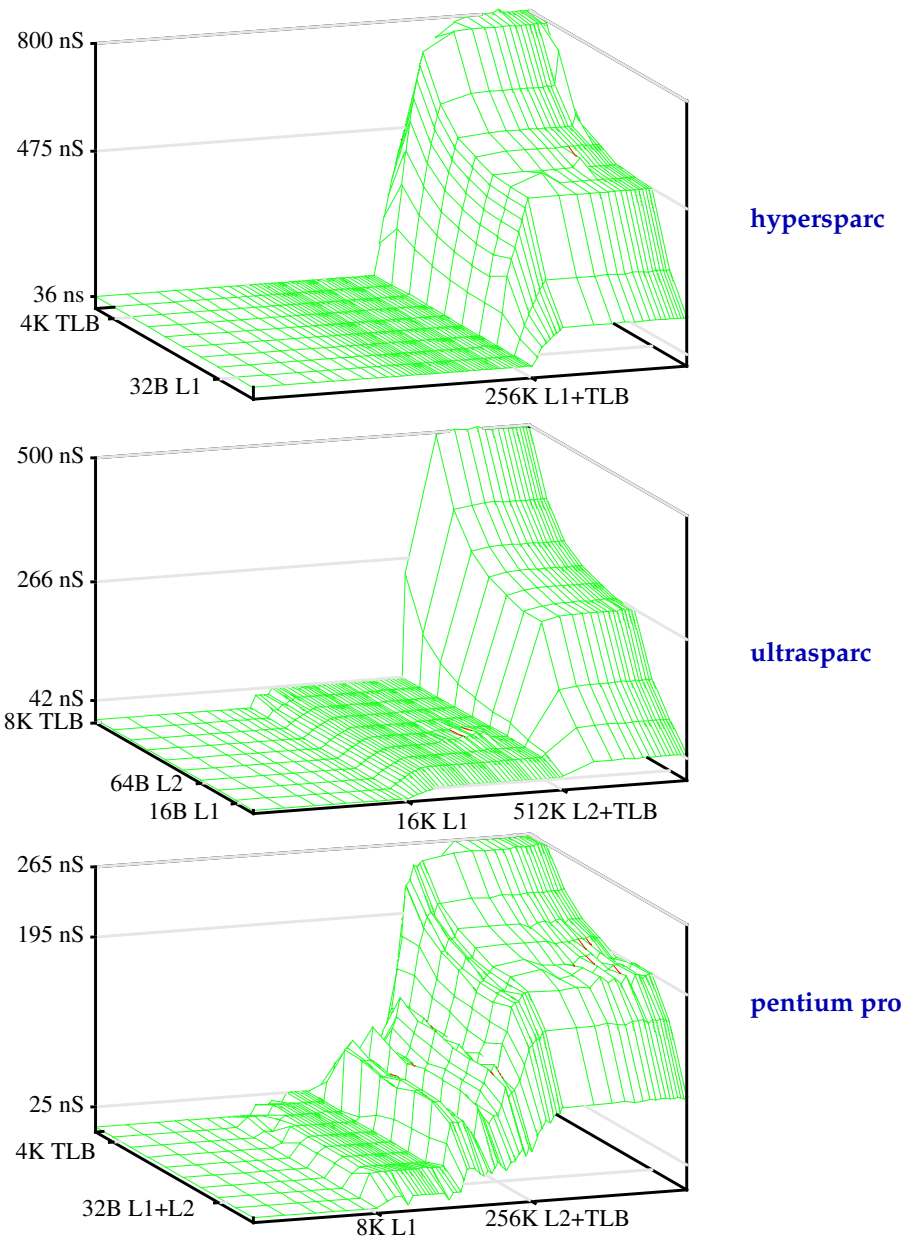


Figure 4: Average latency (vertical) for increasing array size (left to right) and stride. Size and stride are in bytes, log scale.

cessors [56], for example, make it possible to turn off caching on a per-page basis and to access cache memory directly, but to date these operations require supervisor mode and are not given operating system support.)

Cache terminology

The *placement policy* describes the heuristics used to associate blocks with regions of memory. This is usually broken down into the *associativity* and the *replacement policy*. The associativity describes the restriction that the hardware places on which address ranges can be copied into which cache hardware block. *Fully associative* caches can place any (aligned) range in any block; *direct-mapped* caches can place a given region in only a single block; and *n-way associative* caches can place a given region in one of n blocks. Caches may be indexed with virtual addresses or physical addresses. In *physically indexed* caches the placement of regions to blocks is affected by the virtual-physical mapping, so their behavior can be dependent on page placement in the operating system [15][48].

The replacement policy must decide which blocks to throw out to make room for more recent data. The region containing data pushed back out to slower memory is a *victim*. The choice of victim ranges from *random replacement* to least-recently-used (*LRU*) replacement. Sometimes blocks are divided into *subblocks*, which allow portions of the block to be left invalid. *Streaming* caches prefetch blocks ahead of the accessed block, either given an explicit stride or just assuming it is small. A set of memory locations which make up most memory accesses is called the *working set*; being able to hold the entire working set in the cache indicates it is being used effectively.

Additional hardware is often used to improve effective latency or bandwidth. *Write buffers* hold outstanding write requests, waiting for opportunities in which a bus would otherwise be idle to commit the write. The write buffer does *write-merging* if it is capable of combining multiple writes to the same memory region without requiring additional transactions with the next level. Some systems augment a low-associativity cache with a small higher-associative *victim cache*. A *multi-port* cache can support multiple simultaneous accesses. A *pipelined* cache can support multiple overlapping transactions, and when misses don't stop other transactions from proceeding, it is *lockup-free*.

Tolerating latency

When reducing the delay is not enough, latency may be masked by performing other operations while memory transactions are pending. The minimum memory system parallelism required to tolerate latency is given by a restatement of Little's Law for queueing systems:

$$parallelism = latency \times bandwidth$$

For example, a processor with a peak bandwidth of 1 GB/second to a memory with a round-trip latency of 100 nS must keep at least 100 bytes worth of memory transactions pending at a time to hope to utilize that bandwidth. Of course, if computation is to be performed on these bytes, the processor must have sufficient internal parallelism to keep up.

In original implementations of the MIPS-I instruction set, the result of a load is not valid for the following instruction, an example of an architectural concession to tolerating latency. MIPS-II added interlocks, removing the necessity of an architected load delay, but still

requiring the compiler to be aware of the delay for instruction scheduling [56]. More recent machines go much further, allowing multiple outstanding loads and reordering of instructions. A variant of advance scheduling of loads is explicit prefetching, in which an instruction orders that a block should be brought into the cache without tying up a register with it. Processors may also speculatively execute code that depends on memory data. Good instruction generation for processors with lockup-free caches, speculative and out-of-order execution and prefetching is difficult [76].

At a coarser grain, latency can be hidden by switching threads of control whenever there is sufficient latency. When network latencies are on the same order as a conventional context switch, this is difficult to make pay off. There are efforts to architect extremely fine-grain context switches in new designs [93] but at the time of this writing this is not widely available, although some architectural support for fast context switching is. For example, the ultrasparc [90] has limited support for using register windows to hold multiple contexts. Multiple contexts can unfortunately result in a larger combined working set, which can reduce cache effectiveness [94].

Vectorization allows a compiler to generate aggregate instructions which operate on large numbers of independent data items. Because they are independent, memory operations can be pipelined to tolerate latency. Vectorization is only useful when the program is written in terms of data parallel operations or the compiler is able transform the original code into data parallel operations. Vector microprocessors are a simpler and less expensive way to obtain parallelism for these vectorizable problems than superscalar techniques [98].

Dealing with limited bandwidth

While either caching or allowing multiple pending memory accesses help with latency, only caching also helps to reduce bandwidth. While latency can be tolerated by allowing outstanding operations at the instruction or thread level, this has the effect of making operations that were latency-bound become bandwidth-bound because the total number of non-local memory accesses remains the same. Techniques of speculative prefetching can aggravate the bandwidth problem by moving extra, irrelevant data across the pin bottleneck. Eventually increased latency due to insufficient bandwidth must become a barrier to performance [21].

Without a revolution in interconnect technology, bandwidth can only be addressed by transferring less data across chip boundaries. This requires increasing the size of on-chip memory or managing it more effectively. Even caches with perfect use of on-chip memory will still require compulsory memory accesses due to I/O and interprocessor communication, so systems that can tolerate latency by clever caching are also eventually bound by bandwidth [103].

Supercomputers historically solve the bandwidth problem with full crossbar interconnects, many banks of fast RAM, and other aggressive, expensive solutions. The bottleneck may be avoided in the cheaper microprocessor market by finding a way to move more memory onto the processor chip itself, perhaps by integrating DRAM. This will improve

the applications which can achieve high performance because they fit within a chip. However, there will always be the need for I/O, multiprocessor communication, and larger working sets than will fit within a single chip.

A radical proposal for improving off-chip bandwidth is compression, increasing effective bandwidth at the expense of extra hardware on the CPU and in RAM. This is already possible at an operating system level for paging [35] but is not presently feasible at a low level. The constant of compression that can be achieved appears to be quite small.

The only feasible bandwidth solution is to place more memory on chip, as in a cache. Caches relieve bandwidth to the extent that locality of access allows it, so the performance of applications which are bandwidth limited and not vectorizable will be directly limited by the locality present. In a study of SPEC benchmarks, Burger [21] found that between one and two orders of magnitude in effective bandwidth could in principle be obtained by more effective use of the cache. Locality is even important for applications that do not exercise interprocessor communication. Each processor of a Cray T3D has a potential read bandwidth of one 64-bit word every four clock cycles, or 320 MB/sec. The actual bandwidth achieved may be as low as 28 MB/sec, with up to 42 instructions in the time it takes to read a single local DRAM location [68].

Multiprocessor issues

The previous sections showed that poor use of on-chip memory has become performance limiting on serial codes. This section describes how multiprocessing and distributed[1] computation affect memory system performance, further increasing the importance of locality.

There are two potentially complementary ways that processors may communicate. One is through hardware emulation of shared memory, and the other is with explicit messages operating as a separate mechanism from memory. Systems are increasingly combining both, with shared memory clustering small numbers of processors that are then linked by a distinct network.

Distributed systems

Low latency networks are now available that make it economically attractive to connect uniprocessors or shared-memory multiprocessors into larger computational systems. These networks can be used to provide a shared memory abstraction using stock hardware

1. In the high performance computing literature, 'distributed' is used to refer to computation which occurs across processing nodes within a single system for the sake of parallelism. In other literature it is sometimes used to indicate computation which occurs on separately administered systems, for example, client-server access to a database. The former meaning is intended in this work; issues of security or fault tolerance are deliberately avoided.

In a similar confusion of terms, 'multiprocessor' is sometimes used to mean cache-coherent shared memory (as opposed to software distributed shared memory). Here the term multiprocessor just means any plurality of processors.

[7][25][57] or extra hardware [61]. There are many commercially available systems now using the clustered approach of small cache coherent multiprocessors connected by a distinct low latency interconnect.

An efficient shared-memory abstraction can be built on top of explicit message-passing hardware with compiler support. The hardware burden can be eased by exploiting compile time knowledge about references [60]. The ultimate performance of such systems is dictated by how much useful knowledge the compiler and runtime can extract about access patterns; the advantage of doing as much as possible at compile time instead of with hardware is that such analysis is essentially free. In the following chapters, networks requiring explicit message operations are considered part of the memory system if these operations are managed by the compiler and/or runtime rather than the programmer.

Cache coherent shared memory

Efficient distributed shared memory is often emulated by allowing ranges of memory to be replicated in the caches of multiple processors. Copying data increases the effective bandwidth to that data, multiplying the bandwidth of a single cache by the number of processors. But when the data is modified, the copies must remain coherent, either by invalidating all but one copy or by speculatively updating all of them. There is a rich literature on ways to implement this [63], with various performance trade-offs [42][80].

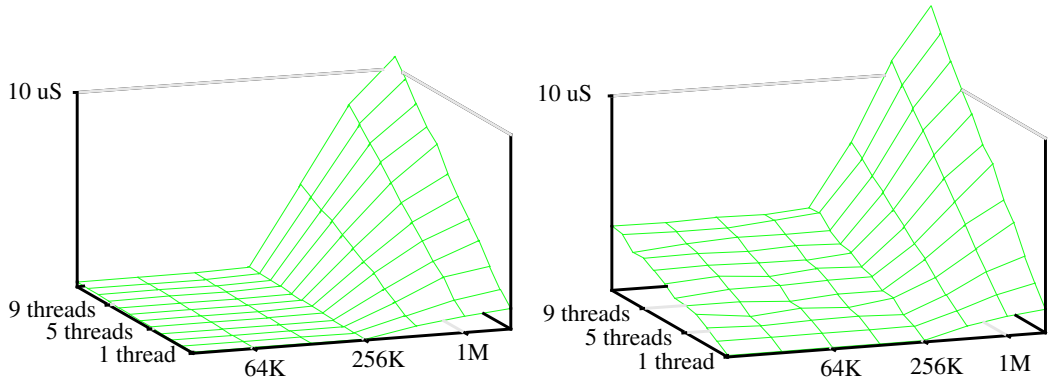


Figure 5: The effects of bus saturation (left) and false sharing (right).

For design simplicity, many commercially available cache coherent systems use a single fast bus connecting a small number of processors, each with one to three levels of local cache. A single shared bus presents a bandwidth bottleneck that is aggravated by adding more processors. Figure 5 demonstrates this for a Sparcstation 10 with four Ross Hypersparc processors on a shared bus. This program forks some number of threads that follow a random pointer tour of a common region of memory of a fixed size in the same manner as [80]. As multiple threads are added that access the same memory, performance roughly scales with the number of executable threads until the working set exceeds what can be

kept in each processor's cache. For the Ross hypersparc modules seen here, this occurs at the cache size of 256KB, after which performance stops scaling with the number of threads because the shared bus is saturated.

There are theoretical results that many algorithms are asymptotically limited by total bandwidth rather than parallelism. In particular, it has been shown that for a sufficiently large fixed problem size, sorting is limited only by available bandwith, not by the number of processors - an infinite number of processors will not help [1].

To eliminate this bottleneck, shared memory is better implemented using a general network, so that bandwidth scales with the number of processors. On such a machine, the sharp knee seen as size proceeds past 256K would still appear, but overall performance would at least be able to scale with more parallelism.

In addition to intrinsic bandwidth demanded by the application, cache coherent machines can suffer from additional memory traffic caused by *false sharing*. This occurs when one processor modifies a location which is not subsequently read by another processor, but still causes an invalidation or update message because it accesses the same cache block. On the right of Figure 5 is a timing graph for the same system, but this time touching each location once during the tour. No thread ever writes a word of memory touched by another thread, but the system is now unable to scale with the number of processors because of the traffic false sharing induces. Although this result was obtained on a shared bus machine, the same behavior would be seen with any kind of interconnect. The congested resource is not any piece of hardware, but a logical span of addresses. The only solution here is to attempt to rewrite the application so that memory written by multiple processors does not reside on the same cache line; this requires that the programmer, compiler, and/or runtime anticipate the hardware caching policies and avoid abusing them.

A bandwidth comparison

Figure 6 plots peak 64-bit floating-point performance against bandwidth as measured by the STREAMS triad microbenchmark [67]. This Fortran microbenchmark computes $a_i \leftarrow b_i + s*c_i$ for arrays larger than the cache size of the machine being tested, and the code is structured so that data re-use is not possible. This tests the memory system's effective throughput. Each point in the figure represents a multiprocessor announced between 1992 through 1996; these are divided into shared memory (cache coherent), distributed memory, and vector machines. Since the triad microbenchmark is representative of long vector operations, it can be expected that vector machines will do well. The feature of greatest interest is whether the point is above or below the line; machines falling below the line are limited by bandwidth to memory rather than floating point performance.

The shared memory machines all have less available bandwidth than most vector or distributed memory machines. The highest performance shared memory machines come nowhere close to the highest performance distributed or vector machines. The line represents the minimum bandwidth needed to sustain the vendor's claimed peak floating-point rate. In contrast to shared memory machines, most vector machines fall near the line; they have the best balance of memory system to floating point for this benchmark. (The one distrib-

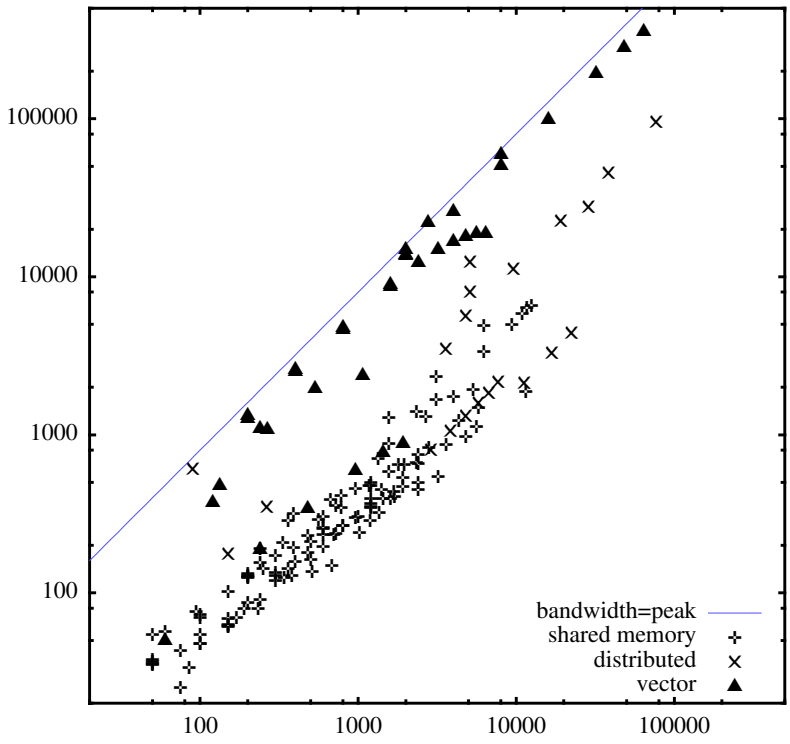


Figure 6: Measured bandwidth (vertical in MB/s) vs. claimed peak Mflops (horizontal) for the STREAM triad benchmark (data from [67]).

uted outlier near the line represents a single processor Meiko CS-2, a hybrid system which has vector hardware within each node, but uses distributed explicit messaging between nodes. With only one node, this could be considered a vector machine.)

Consistency models

The preceding sections outlined a few reasons that multiprocessors tend to use caches less effectively than uniprocessors. Luckily, multiprocessors also present an opportunity to ameliorate the effects of cache misses not available to uniprocessors.

When there is only a single thread, memory is a simple thing: a memory cell or variable always has the last value assigned to it. However, when there is more than one thread of control and no explicit synchronization, the order in which a thread sees the writes of another thread isn't well defined - depending on thread scheduling, one thread may race ahead of the other and write values that are subsequently read, or it may instead be forced to idle. Because programmers can't be assured of any particular order that the threads will execute, they also aren't guaranteed an ordering on memory accesses.

Conventional languages don't have threads, so the behavior of memory isn't hampered by legacy semantics. The nondeterminism of thread scheduling makes it possible to define new semantics of memory access. It is advantageous to give the programmer as few guarantees about memory order as possible so as to give the implementation the most flexibility, enabling higher performance. The guarantees a multithreaded system makes to the programmer about the behavior of memory are called the *memory consistency* model. The consistency model applies to both message passing and cache coherent machines. The implementation of the model may be in hardware, controlled in software by the compiler and runtime, or both.

The most intuitive consistency model is *sequential consistency*, in which all memory operations are guaranteed to appear to occur in some order consistent with the execution of each thread. The implication for a cache coherent system is that each memory operation - a read or a write - must appear to complete before another can be started. This model is the easiest to reason about.

A common alternative is *processor consistency* (used by Intel multiprocessors [53] and others). Write buffers were introduced on page 19 as a hardware technique to allow other operations to continue while writes complete. For uniprocessors, the programmer can't tell whether or not there is a write buffer (other than timing): when a read occurs to a location waiting to be written to, the write buffer steps in and provides the newest value. For multiprocessors, writes may sit in the write buffer and not be immediately visible outside the processor. Processor consistency guarantees only that the order of writes from a single processor will always appear in order, not that the interleaved order of writes from different processors appear the same. This allows the write buffers to be used; under sequential consistency they could not.

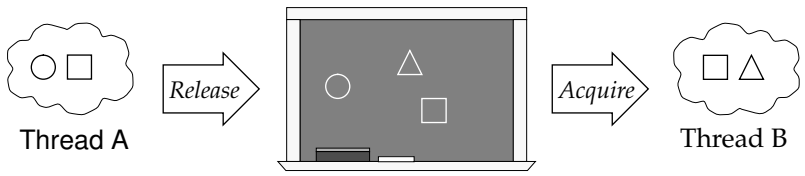


Figure 7: The acquire-release consistency model. Another processor or thread is not guaranteed to observe changes until the writer has performed a *release* and the reader has performed an *acquire*.

Processor consistency allows processors to continue working without waiting for writes, but what about reads? Program order between threads is defined only by events which require synchronization, such as waiting for a barrier; this suggests having synchronization actions play a role in the behavior of memory. A model offering still weaker guarantees than processor consistency is *release consistency*, which requires writers to explicitly *release* their updates to memory and readers to explicitly *acquire* these changes before guaranteeing they will be observed (figure 7). These releases and acquires are usually automatically associated with synchronization actions in a program by the compiler inserting special memory instructions at these points. For example, if releases occur when threads enter a

barrier and acquires occur when they leave, all threads will see all changes by other threads at the time they resume execution. This allows processors to continue working through both reads and writes - until a synchronization event occurs.

Weakening the consistency model in these ways allows processors to keep working after initiating a memory operation without always waiting for it to finish. This can improve performance; for example, Sather uses a weakened consistency model (page 122) similar to release consistency that has be used to enable significant optimizations on distributed systems [42]. The down side is that programmers may only use explicit synchronization to control program order between threads. Memory order may behave in a nonintuitive way, making program behavior more difficult to reason about. The next chapter will describe other serious software engineering difficulties these weaker models create.

SOFTWARE

Cache hardware tries to exploit regularities in the pattern of memory accesses to reduce latency and improve effective bandwidth. Now we examine ways that software can be constructed to capitalize on the hardware by offering as much exploitable regularity as possible.

Fetching of entire cache blocks surrounding an accessed location is a form of speculative prefetching based on assumptions of spatial locality. This trades effective bandwidth for latency toleration. Most instruction sets provide some support for software controlled speculative prefetching as well; for example, non-faulting load instructions and loads that bring data into the cache without binding the result to a register [90]. Insertion of these instructions may be considered a code generation problem, often only useful in combination with software pipelining. These techniques tend to consume bandwidth. Since bandwidth is expected to become a performance limiter, this chapter does not concern itself directly with prefetching transformations, instead concentrating on reducing required bandwidth by reducing cache misses.

Why misses happen

On current processors the speculative prefetching of entire cache blocks by hardware is nearly unavoidable. Although prefetching negatively affects bandwidth, the degree to which it does so is affected by the spatial organization of data. The ratio of traffic passing through a cache to the traffic it handles provides an accurate measure of how on-chip memories affect bandwidth [21]. Because this work does not speculate on alternate cache configurations, the cache hit rate will be considered an adequate measure of bandwidth reduction for a given block size with a direct impact on application performance. We now examine what causes caches to miss.

A taxonomy of misses

Misses at any level have elsewhere been categorized as *compulsory*, *capacity*, or *conflict* [62]. Compulsory misses occur when accessing data which was not previously in the cache. Capacity misses are a result of accessing more data than can fit in the cache. Conflict misses occur when the placement policy victimizes blocks. While it has proved useful for uniprocessors, this taxonomy has two unacceptable inadequacies: it doesn't address misses that result from cache coherence interactions, and conflict misses are given a clumsy definition in terms of LRU replacement. A much cleaner definition may be made in terms of a (fictional) optimal replacement.

Here an alternate taxonomy is presented:

> *Compulsory misses* are a result of accessing data that is not in the cache. This means accesses to locations when they are visited for the first time, or I/O.
>
> *Coherence misses* arise from a processor reading a location which would have been in the cache, but was invalidated by another processor writing to it.
>
> *Capacity misses* are those that a perfect cache could not have avoided. A perfect cache has the same capacity and block size, but is fully associative and has an optimal replacement policy (the block is victimized which is needed furthest in the future). Such misses cannot be blamed on associativity or replacement policy.
>
> *False sharing misses* occur when a block is invalidated by another processor which did not write to the location being accessed. On coarse-grain SPLASH programs, misses due to false sharing comprise 40 to 90 percent of all misses for a variety of block sizes [54].
>
> C*onflict misses* are everything else.

This treatment of misses fails to model bandwidth issues on systems that use speculative update rather than invalidation protocols, because formally no coherence or false sharing misses occur. In practice, the bandwidth consumed by an invalidation protocol tends to be similar to that of an update protocol [65], although latencies may be strikingly different.

Reducing misses

There are two ways to reduce misses by program transformation: changing where data is placed, and changing when it is accessed. Here a list of policies is presented, each of which may have benefits at various levels of the memory system.

> *Alignment* is arranging of data items which are used together to stay within block boundaries. This may mean padding objects to a multiple of the block size and allocating aligned to block boundaries. Alignment may reduce compulsory misses by prefetching other object fields. Capacity and conflict misses may be re-

duced by always loading the fewest number of blocks needed to contain the object.

Clustering is arranging for data which is used closely in time, or by the same thread(s), to be allocated within the same block. Clustering is implicit in the layout of objects as contiguous fields. Small objects may be clustered on the same cache line, and related objects may also be usefully placed on the same page or node. Clustering may reduce compulsory misses by prefetching objects before they are used. It may reduce coherence traffic by placing objects within the same processor. Capacity misses may be reduced by having more relevant data in each block, and conflict misses are reduced for the same reason - there are more free blocks around. 'Locality' is usually taken to mean clustering.

Zoning is the complement to clustering: deliberately placing data that is not used closely in time or by the same threads in different blocks. Zoning has largely been ignored in uniprocessor cache analyses, but can impact performance on multiprocessors, reducing false sharing by not allowing objects used by different processors to reside on the same block.

Reordering refers to code transformations which change the order in which data is accessed, such as loop fusion, interchange, and blocking. These transformations are well understood [11] and unassisted compiler transformations have been very successful at reducing capacity misses on some codes, especially dense linear algebra.

Individual transformations may help implement more than one policy; for instance, padding may be used to simultaneously implement alignment and zoning. It should be understood that misses are being used in a way not restricted to cache coherent machines. For example, a Cray T3D allows reads and writes from remote nodes, but does not cache these. Although there is no cache involved, remote accesses are still misses in that they contribute to required bandwidth and can be avoided by better placement of data. On systems with explicit messaging, 'misses' can be interpreted as the overhead incurred by the compiled code and runtime to deal with messages.

There is plenty of room for improvement in existing systems by applying these policies: in a trace-driven study, Markatos [65] found that effective memory access cost on a hypothetical coherent machine could be reduced by one to two orders of magnitude on scientific codes by optimal placing of data through off-line analysis. Good placement was much more important than the details of the coherence protocol.

Locality maintenance

This section describes conventional ways that locality can be improved by compiler optimizations, feedback guided manual code transformation, or adaptation at runtime.

Compiling for locality

Many compiler optimizations, such as register allocation, redundancy elimination and loop invariant motion may be viewed as potential locality improving transformations. They can reduce the number of memory accesses and non-register variables. Because these optimizations are well understood they are not considered further here.

There is a large literature on improving the locality of loops over array data, which will only be briefly mentioned here. Nested loops can be interchanged to decrease stride. Skewing changes the array layout to improve stride for affine access patterns of multidimensional arrays. Blocking or tiling performs matrix operations by dividing them into operations on subblocks, each of which may fit in the cache. The granularity of computation can be improved to enhance temporal locality by combining operations from independent iterations of a loop into composite operations. These operations must be combined in a common framework that is informed of the target memory system characteristics for best effect [11, 23]. In addition to blocking transformations already discussed, on some machines the predictability of strided access can be exploited by the compiler to take advantage of read-ahead prefetching, page-mode RAM, memory interleaving, and special stream hardware.

Code transformations have also been applied to linked-list traversal, which are frequent in programs written in functional and logic languages. When the last word of an object points to the following object, the pointer may be omitted by placing the objects together. More generally, pointer data structures can often be replaced by arrays. This change in representation improves locality by reducing space consumed and clustering related items. To get the best effect, loops have to be unrolled and it must be possible to rule out aliasing [84]. Memory management and related cache behavior of the application also substantially affect the results [9]. It is unclear how far automated analysis can be taken with more complicated data structures although some analysis is possible in simple recursive structures [79].

Because compiler optimizations often depend on non-local dependency analysis, they violate compositionality of the performance model. For example, a loop transformation may be possible if the compiler can show that two pointers can't alias the same region of memory. Proving whether they do so may require analyzing all code reachable from within the loop. This can cause minor code changes to have performance consequences that propagate to seemingly unrelated parts of the application.

For uniprocessor systems, failing to optimize will generally bring a small constant factor of performance loss. On multiprocessors, however, the wasted time may not only be multiplied by the number of processors, but coherence effects may introduce very different asymptotic behavior. An application that scales linearly with the number of processors may change into one with no scaling at all because of placement decisions that result from subtleties of optimization that are not apparent to the programmer.

Manual transformation

Manual code transformations can be effective when automated techniques fail [62]. Identifying which data structures are responsible for poor cache behavior requires knowing their layout in memory, the processor(s) reference pattern and details of the architecture and coherency protocol. For this reason good source-level tools are essential, but at this time such tools are rare.

Jeremiassen [54] found that manual restructuring of data placement could reduce false sharing misses by up to 75% on sample coarse-grain programs. Automated code transformation was generally more successful, eliminating on average 64% of false sharing misses and up to 90% for programs that had been manually transformed. The most successful optimization was simply padding locks so that modifications to critical region variables did not cause loss of exclusive block ownership.

Locality at runtime

There are off-line, runtime approaches to obtaining locality. For example, solving differential equations on unstructured meshes is frequently handled by a preliminary phase which determines how data structures will be mapped and the scheduling of work and data movement which takes place in a subsequent execution phase. The preliminary phase may either be done off-line or considered part of the execution [91]. This approach is sensible when the cost of the preliminary phase is small compared to the overall computation.

However, there are many problems in which the data structures, communication, or scheduling cannot be reasonably predicted in a separate phase. For such irregular problems [101] it is necessary to actively maintain locality, allowing parts of the computation to proceed before an optimal placement or schedule can be obtained. Here we examine locality solutions for irregular problems that cannot be solved adequately by compilation or preliminary analysis.

Because the placement of objects in memory directly affects cache behavior, there has been investigation into the effects of memory management algorithms on locality. Locality effects due to memory management are often hard to distinguish from raw application locality; when they can be, the results are hard to translate between languages, compilers, and memory systems [102]. There is consensus on two points: memory management algorithms themselves exhibit poor locality, and relocating garbage collection can improve locality under some conditions.

Grunwald [45] found that the cache locality of common malloc/free memory allocators can be quite poor, generally trading off better fragmentation policy for poorer reference locality in the allocation routines. In a later study of conservative garbage collection Zorn [106] found that there was often an implicit cost to explicit memory management. For example, reference counting is extremely common in C++ programs. However, updating reference counts stored with the referenced objects on every pointer assignment touches the

cache blocks of those objects, on which data may not even be referenced otherwise. It was concluded that because it may improve the reference locality of the application, conservative garbage collection should be considered even when performance is the primary goal.

Some garbage collectors are able to relocate objects after allocation; these can improve locality by clustering objects that refer to each other. Another study of the locality effects of garbage collection showed that copying collection can improve the application locality in large Lisp systems. However, for direct-mapped caches, mark-and-sweep collection itself showed much better locality [105]. Copying garbage collected systems with high allocation rates appear very sensitive to whether cache lines are forced to be allocated on the first write to a non-allocated block, because most writes occur to untouched space for which prefetching brings no benefit.

Bonwick [17] found that systematic alignment of Unix kernel data structures to powers of two could reduce cache effectiveness due to associativity. To resolve this, arrays of objects were artificially padded to spread out their alignment. This demonstrates one way in which the alignment policy (page 27) can fail. Austin [10] explored the use of profiling feedback to divide allocation sites into sets, with demonstrable reduction in conflict misses.

Operating system support for handling of process scheduling, paging and TLB replacement policies has been extensively studied [28][95]. Two heuristics that improve locality are giving a process affinity for the processor where it last ran, and increasing the duration of the time slice itself. Burger [19] examined the impact of conventional page fault handling on massive multiprocessors, concluding that such systems require gang scheduling and other modifications. Automatic page migration schemes have been proposed which try to migrate pages to processors incurring cache misses on that page, including a variety of software-only distributed shared memory implementations [7]. Cao [22] describes the design of a file system that integrates application-controlled caching, prefetching, and disk scheduling.

Performance Models

The previous chapter, **Memory Systems and Locality (page 14)**, reviewed techniques that can be used to obtain locality without changing the way programmers have to think about the systems they build. While successful for particular classes of applications and specific hardware, these automatic techniques have not been able to obtain optimal performance in general. This chapter reviews *performance models*: ways that programmers can reason about performance when trying to build systems which meet the above goals. Unlike the techniques of the last chapter, using a performance model to obtain locality sometimes requires programmers to think in nontraditional ways, making it possible to build systems which are not oblivious to the performance characteristics of the underlying hardware.

This chapter begins by reviewing common performance models that are in practical use as well as more academic models that have been proposed. Recall that the goal of this work is to enable high performance and programming in the large at the same time. Requirements mentioned in the introduction (page 8) included:

1. **High performance** - performance must scale with larger/faster hardware and degrade gracefully under additional software load,
2. **Portability** - correctness and performance must persist when hardware changes,
3. **Modularity** - correctness must persist and performance must be predictable when independently written software modules are composed,
4. **Usability** - structure, coding techniques, and implicit assumptions should be clear and available so that code can be reused, extended, and maintained. Applications should be not restricted to narrow domains such as scientific computing.

None of the models meet all of the goals above, but two models here are of particular interest because they pave the way for a new model. The low-level **Parallel Memory Hierarchy model (page 42)** allows accurate description of hardware by distinguishing not only clusters, but hardware units at other levels of the memory hierarchy as well. The **Sather 1.1 distributed extension (page 47)** is a higher-level model which presents a view of hardware as a collection of clusters of processors; all memory accesses within a cluster are inexpensive, while memory accesses between clusters are expensive. The next chapter, **Zones (page 50)**, describes a new performance model which combines the high-level convenience of the Sather distributed extension with the expressive hierarchical approach of the Parallel Memory Hierachy model.

PRIOR MODELS

This section reviews prior work on performance models by classifying each as implicit, explicit, or annotative. The most common performance models for general purpose programming are *implicit* - that is, the factors that most affect performance are not reflected as features in the language and programming environment. Instead, the programmer is taught how to infer performance effects based on knowledge of how compilers, runtimes, and hardware operate. Other performance models are *explicit*, requiring message sends, absolute placement or restricting the available kinds of synchronization. Recent languages have had *annotative* locality models which allow the expression of locality to be independent of the correctness of code.

This section covers practical models actually applicable to large software systems. A substantial and largely disjoint body of literature covers mathematical models used for complexity analysis of parallel algorithms (eg. PRAM, LogP [33]). These performance models are of greater theoretical than practical interest so are not discussed here.

Implicit models

Caches and coherent shared memory are a way to lighten the programmer's burden by doing extra work in hardware. Cache hardware has evolved to make the best use of the kind of code that compilers generate (i.e. a specialized stack) and compilers have co-evolved to generate the best code for existing caches. As a result, there are performance conventions that are implicit. Although cache systems are programmed as if they have a uniform flat address space, in practice programmers have learned programming idioms that result in improved memory system performance. The performance expected of these idioms have becomes no less important than any formal language specification.

Common assumptions

Expert programmers have learned to rely on automatic local register allocation, invariant code motion, constant propagation and common subexpression elimination. Programming is extremely tedious without such optimizations and reduces the usefulness of common practices such as macro expansion. Similarly, many languages require garbage collection for applications to not overconsume memory but do not require it formally since, by definition, it has no semantic consequences.

Compiler loop optimizations such as unrolling cannot always be relied on, but there are other characteristics of code generation for arrays that are trusted. Compilers are expected to allocate arrays in contiguous memory. On a cache coherent machine, this means that generally, locality of access in the array indices will translate into fewer cache misses. It is possible, for example, to perform manual blocking of matrix operations without knowing the exact cache organization of the target. Going from a general notion of improved locality to optimal blocking for a given memory system is non-trivial [30].

Compilers are similarly expected to lay out object fields in contiguous memory[2]. The implication is that once a field is accessed, subsequent accesses to other fields of the same object provide locality. In [62], for example, one beneficial code transformation is the combination of multiple arrays of related data into a single array of structures. Similarly, local variables (and often arguments) of a single method can be expected to occupy a contiguous stack frame.

Thread abstractions can usually be assumed to provide locality within threads, but not between them. For example, two threads operating on an array may each benefit from the locality of their own accesses to the array, but suffer when each accesses the same area at the same time. This is an example of implicit zoning. On some systems the performance conflict between threads has erupted into relaxed memory semantics, with explicit consistency models that require programmer attention to points of synchronization [42, 59].

Multiple threads make the best use of the cache if the time a thread runs before blocking is large relative to the penalty of reloading the cache. The interleaved execution of many threads effectively combines their working sets, reducing cache effectiveness. It is important to schedule threads to take advantage of cache affinity [12].

Hill and Larus [50] describe four abstract models of caching useful for programmers who know little or nothing about hardware and are programming with explicit placement of threads among processors:

1. *No caches* - non-local communication can only be reduced by eliminating memory references, such as by keeping results in registers. (Note that this relies on implicit models expected of compilers to be able to reason about what will be in registers.)
2. *Infinite word caches* - Block size is a single datum, and there are only compulsory and coherence misses. Once a location is read it remains local until another processor modifies it.
3. *Infinite block caches* - Block size is known but there are an infinite number of blocks; there can be false sharing misses.
4. *Finite block caches* - Block size and cache size is known, so there can be capacity misses as well. Optimizations for fixed cache size must be stressed less than for uniprocessors, since changes that reduce cache misses can increase false sharing.

The problem with implicit models is that they are a moving target. As compilers and memory systems have changed, programmers expectations have had to as well.

What is wrong with malloc and free?

Implicit performance models that depend on locality are influenced by memory management. Most languages and systems provide a means to explicitly allocate and deallocate memory; in C, this is done with the routines malloc and free. By any name, the conven-

2. ...although there are major differences between languages in treating layout and object-oriented inheritance.

tional interface to object allocation fails to address modular and high-performance goals. Malloc encourages thinking of memory as flat, and free introduces an obscure but severe problem in multiprocessor systems.

The previous chapter detailed ways that memory isn't really flat - where data is placed does affect how long it takes to access. Conventional allocation doesn't allow the programmer to express that objects should be kept together or apart. Even programmers that know about the memory system can't use malloc to their advantage, because it doesn't allow alignment to cache lines or pages to be expressed. Similarly, it doesn't permit the expression of affinity between threads and objects. While alternative interfaces to allocation are sometimes available (eg. memalign, which extends malloc with an alignment requirement), they fail to be portable across memory systems.

In addition to these problems, explicit deallocation can cause nonintuitive failures when combined with weakened consistency models. Suppose the code of the left column of table 1 runs on a system with either sequential or release consistency. A programmer might reasonably conclude that the only possible simultaneous values of g and g.field are (a, 1), (b, 2) and (c, 3). The race condition between the assignment of g and the observance of its value should not affect this: each assignment to the global g is atomic, so even if later values of g aren't observed, at the very least the values of g.field should still always be consistent with the value of g.

	Code	Sequential consistency		Release consistency	
		g ==	**g.field ==**	**g ==**	**g.field ==**
	object a, b, c; *-- local variables*				
	global object g; *-- a global variable*				
	a = malloc(...);				
	a.field = 1;				
	g = a;				
	... another thread is forked here ...				
	b = malloc(...);	a	1	a	1
	b.field = 2;	a	1	a	1
	g = b;	b	2	b	2
✕	free(a);	b	2	b	2
	c = malloc(...);	b	2	b	2
★	c.field = 3;	b	2	b	2
	g = c;	c	3	c	**1**

Table 1: Stale values appearing under release consistency.

Under sequential consistency, other threads observing the global state of memory will observe the values in the second two columns. Under release consistency, the memory occupied by object a may be reclaimed by the explicit deallocation (indicated by the ✕) and reused for object c. Since this memory may be in another processor's cache and there has been no acquire to flush it, this could lead to another thread observing the old value for g.field - the value belonging to a, an object that no longer exists!

How can this problem with release consistency be avoided?

- *Have a compiler insert acquires.* If the observing thread performs an acquire before accessing g.field, the newest value would be seen instead of the stale one. Solving the problem this way would require an acquire before every dereference of a pointer to global memory that might have been reclaimed by a deallocation. While conceivable, this will result in a potential acquire inserted before every read unless some kind of aggressive global analysis is used, defeating any performance advantage of release consistency. Even if compiler analysis can help, programs could be slowed down by unbounded factors by adding an unfortunately placed explicit deallocation.
- *Give* free *a semantics of flushing remote copies.* Instead of requiring the reader to flush their copies, perhaps the processor reusing a deallocated region could somehow flush all remote copies. A problem with this is that there isn't any way to do it - acquire and release events don't affect remote processors. If there is some way of flushing remote caches, doing so for every deallocation could get expensive.
- *Don't use explicit deallocation.* This whole problem is caused by not having a one-to-one relationship between program variables and memory - when memory is reclaimed, old program variables might be seen. If the deallocation marked by X were simply removed, the problem would go away because memory would never represent more than one program variable. But most programs do need reclaimation of memory.

A fourth alternative is to combine the second and third options with garbage collection.

Garbage collection

Widely used imperative languages such as C, C++ and Fortran 90 require the programmer to explicitly manage dynamically allocated storage. Good programming practice suggests making procedures locally responsible for related data structures. Unfortunately, memory management issues often cut across natural abstraction boundaries, making interfaces unpleasant in having to include low level deallocation issues.

A programmer doing explicit storage management may mistakenly free an object while it is still being referenced, leading to a *dangling reference*. A programmer may also forget to free memory even when there are no references to it, leading to *memory leaks*. Both mistakes can be very hard to detect because their malevolent consequences may show up unpredictably, far away in space and time from the actual origin of the error; perhaps only with particular input data or on particular platforms. These problems are severe enough that a small industry has arisen to provide tools to help C and C++ programmers find such bugs (Purify). C++ classes often use manual reference counting techniques to provide a crude form of reclamation, but these fail with reference graphs containing cycles. They also introduce substantial performance hits by greatly raising the cost of pointer assignments and decimating locality of reference.

For these reasons, many languages are *garbage collected*, so programmers never have to free memory explicitly. The runtime system does so automatically when it can be proven to be safe. With explicit deallocation this work is done by the programmer, although such explicit storage management often has little effect on performance [106]; it is rarely worth the effort or complexity.

There are good software engineering reasons to use garbage collection, but it also can provide a solution to the stale-value dilemma of the last section. Instead of flushing remote copies for every call to free, garbage collection can flush remote cached copies of memory that it reclaims all at once. This isn't likely to have the same performance problem as associating flushes with every free, because garbage collection is occasional and coordinated. Remote copies only have to be flushed once per garbage collect cycle, not once per object freed.

Garbage collection does not entirely prohibit explicit deallocation; the programmer can be allowed to manually deallocate objects, letting the garbage collector handle the remainder. This allows garbage collected languages to be used even given tight constraints such as real time systems. It is also possible to treat such explicit deallocations as assertions so that the system will catch dangling references from manual deallocation before any harm can be done. When such checking is done, the program cannot crash disasterously or mysteriously. All sources of errors that cause crashes are either eliminated at compile-time or funneled into narrow circumstances (such as accessing beyond array bounds) that are found at runtime precisely at the source of the error. This approach is used by the Sather language.

Various types of garbage collection carry their own implicit models of locality. For example, the additional memory system load of garbage collection can reduce the apparent benefit of modifications to applications to improve their internal locality. Copying garbage collection may encourage programmers to believe that objects that refer to one another are more likely to be close in the heap and thus provide a measure of locality [49].

Explicit models

The implicit performance models discussed in the previous section require that the programmer think on two levels. One level is essentially syntactic - constructs such as routines, variables, and objects that make up the language used to express the program. At the same time, the performance-minded programmer must understand how these constructs map into the time and space of hardware - the performance semantics. Hiding the performance from the programmer in this way is both a weakness and a strength. Implicit models make attaining performance harder by not allowing expression of performance characteristics of the code directly, but they allow portability by not commiting to the performance characteristics of a particular platform.

In contrast, explicit models deliberately expose more of the performance characteristics of the hardware to the programmer. This can be done by having the programmer specify the dependency structure of their code (explicit dependence), or by having the programmer use constructs that directly correspond in time and space to the executing hardware entities (explicit communication).

Explicit (in)dependence models

Rather than requiring explicit placement of each data item, a variety of systems attempt to automate placement based on explicit dependence information extracted by the compiler or provided by the programmer. This is done at a fine grain with data parallelism, and at a coarse grain with explicit dependence.

Data parallelism in the form of vector operations was already discussed in the context of compiler optimizations. Vectors make explicit the independence of data elements, allowing benefits of parallelism and pipelining with little hardware complexity. For programs composed of vector operations, vector microprocessors may provide an inexpensive way around latency and bandwidth issues without resorting to design-intensive out-of-order superscalar processors with massive on-chip cache [98].

Data parallelism allows compiling away the need for synchronization checks at runtime. In vector operations, there is no communication between individual element operations, but with compiler support this can be relaxed, allowing more general code than vector processing. For example, Modula-2* [75] offers parallel constructs which synchronize on every statement, as well as constructs which allow decoupled, asynchronous execution of statements in SPMD fashion.

Jade [78, 81] is a coarse-grain, deterministic system which attempts to schedule tasks on the same processor as other tasks that accessed some of the same objects. Jade was built on top of SAM [82], a runtime that does implicit software caching of remote values, enabled by immutable (versioned) object semantics.

Data parallelism is appropriate for problems which can be naturally decomposed into independent operations. Most scientific programming is of this form, but there are other important domains - such as compiling, simulation, and some database management - which do not decompose naturally in this way.

Explicit communication models

There are a number of libraries that may be used with C or C++, providing a shared address space and synchronizations [Presto, ANL macros]. A typical runtime library consists of multiple light-weight tasks that execute in the same address space, communicate through shared memory, and synchronize through constructs such as locks, barriers, and atomic regions. These libraries tend to not handle locality well. In the ANL macros, the only support provided for locality are primitives to lock a process to a processor or allocate data from a processor.

Non-coherent systems require explicit message libraries such as PVM [41], MPI [34] and AM [36]. Rather than a shared address space, these libraries provide primitives for sending and receiving messages with a variety of related synchronization. Additional facilities are required for placing and distributing code and data and managing I/O. Libraries can be built on top of message passing to encapsulate regular communication and synchronization patterns (eg. BSP [97]). It is desirable to provide a hardware abstraction such that threads and synchronization do not directly interfere with the expression of communication. Ideally, synchronization should be orthogonal to placement.

Instead of programming directly using these libraries, some languages use them as a compilation target and reflect locality in the type system. Split-C [32] is typical, distinguishing local pointers and far pointers to data on remote nodes. Attempts have also been made to model side-effects directly using types [43], though with little practical application.

CRL [55] and Shared Regions [40] allow the binding of variables to regions with explicit control of consistency; explicit actions are required to bracket read or read/write access.

Explicit messages often require the programmer to violate encapsulation and compositionality of software modules, so the various forms of explicit communication appear viable only for wizardly high performance programming or simple tasks. These models also assume the memory system has at most three levels relevant to performance - local memory and remote memory and sometimes local cache.

Annotative models

Fortran has a brave tradition of interpreting comments as compiler hints, allowing optimizations that may require more analysis than a compiler is otherwise capable of. Dividing a language into a core and performance-affecting annotations can ease code development, because a correct program can be written before worrying about efficiency. Often there are directives that *can* affect correctness but do not if used in the intended way.

In High Performance Fortran (HPF), a serial program may be annotated with data distribution specifications. A variety of primitives are provided for this including distribution patterns such as interleaved, round-robin, blocked, cyclic, alignment, and dynamic distribution. HPF annotations are used by a compiler to extract and schedule parallelism and communication from a serial Fortran program. Along with concurrent computation primitives, the data distribution annotations may be used by the compiler to partition the program into parallel activities based on an owner-computes rule. Both data and machine are abstracted as multidimensional grids, but mapping is performed at only a single level of the memory system, implicitly depending on the compiler for other locality.

There are a number of object-oriented languages that have annotative constructs for placement. Emerald [52] was designed for distributed environments and has mechanisms for specifying the location of objects. Objects may be fixed to other objects, but if unfixed they may move at the system's discretion. It is possible to query an object's fixed or unfixed sta-

tus, and find out what it is fixed to. Objects may be moved on demand. When objects can be moved on demand, access to them must go through an indirection, which penalizes performance even when objects do not move.

In COOL [29], calling a function prefixed with the keyword parallel forks a thread to execute the function asynchronously. Like many systems, hardware is abstracted at two levels, a local memory with a cache and remote shared memory. COOL uses a form of object-affinity scheduling that runs the thread on the processor whose memory holds the invoked method's object. Optionally, affinity to other objects may be specified. Object affinity co-locates threads with an object. Task affinity schedules threads back-to-back. Simple affinity is a combination of object and thread affinity, co-locating a thread with an object, scheduling all such threads back-to-back. Processor affinity specifies which processor a thread will run on. Objects may be allocated from a particular processor, and migrated on demand, although this moves the entire page with possible unrelated objects as well. A construct allows array distribution. Finally, the current processor holding an object can be queried. Chandra [27] reports moderate success with improving performance of SPLASH benchmarks on the DASH using these affinity annotations. COOL has the liabilities of abstracting hardware at a single level and nonportably involving page size in migration behavior.

Prelude [99] also allows explicit migration of threads and objects, depending on compiler analysis to generate code that is not impacted by indirection between object migration.

Fowler [38] implemented a task-queue with an object-affinity runtime that was able to improve performance of parallel programs at a granularity too fine for thread-based cache affinity to be effective.

The Check-In/Check-Out (CICO) model [51] allows bracketing code with directives that indicate when shared or exclusive access to data begins and ends. This is superficially similar to inserting coherence primitives in software cache-coherent systems such as Shared Regions. Data is expressed in terms of blocks with explicit address ranges, and CICO does not attempt to manage synchronization.

Zhang [104] explored replacing the implicit prefetching of long cache lines with short lines and explicit prefetch control, relying on the compiler and programmer to annotate groups of variables. These variables can then be prefetched together by hardware which recognizes special bits associated with each line.

CASE STUDIES

Models are now examined in greater detail which motivate the zones model presented in the next chapter. Models of register use have arisen through which portable and modular performance has been achieved. The Parallel Memory Hierarchy model is explicit, and important because it allows simultaneous reasoning about multiple levels of the

memory hierarchy. The last model examined here, the Sather 1.1 distributed extension, is high-level, annotative, and distinguished by its careful codesign with a high level language.

Portable performance with registers

Registers make up the first and best understood level of the memory hierarchy. We will now examine the explicit, annotative and implicit models that arose to address the performance issues of registers.

In assembler languages, all register use and memory access is explicit. The precise timings of individual instructions might be abstracted away to allow reimplementations of the instruction set, but a specific timing model is never far away, often right alongside the assembler reference manual. So in assembler, the performance model must be considered explicit.

C has an annotative performance model for registers. Local variables may be declared with the keyword register. The only semantic effect to this annotation is that it allows the compiler to complain if the programmer attempts to compute the address of the variable. C was designed at a time when many compilers weren't able to do a good job of register allocation, so this hint by the programmer could be a large win for carefully designed inner loop code.

More recent compilers can do a better job of register use than programmers can with reasonable effort. Most compilers now ignore the register hint and do register assignment as if it weren't there, because it turns out not to matter. Programmers who understand that this is how modern systems work are using a performance model that is implicit.

The model of register use can make a big difference. For example, a programmer may consider unrolling a loop to allow latency tolerance through improved instruction scheduling in larger instruction blocks. There are also other advantages to unrolling loops, such as to amortize the per-iteration overhead over more than one iteration. The number of iterations to unroll depends strongly on the number of registers available; if the loop is unrolled too few times, registers may remain unused and cause memory latency to be a bottleneck. If the loop is unrolled too many times, more registers may be required than exist and register spilling - moving values back and forth between registers and memory to compensate for too few registers - may occur. That makes a tradeoff between having the limited resource be memory latency of the number of registers. This relationship is illustrated in figure 8; the exact curve would depend on many things, such as the quality of the compiler, icache structure, and what the operations in the loop body are. Although more inflection points may appear due to other resource constraints, the overall shape will be similar.

Given that the number of registers and other factors influence the optimal unrolling, it is clear that the very highest performance code can't be made portable without information about the hardware. And yet, programmers do manage to write high quality portable inner loops, by using compilers that are capable of making the right decision and carrying out

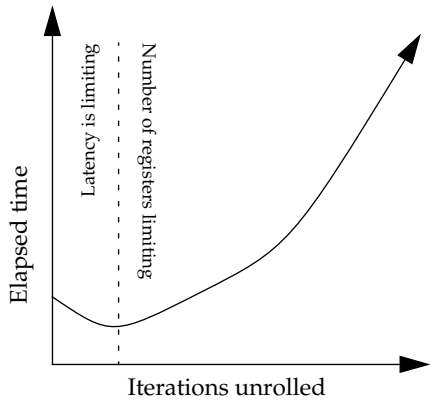


Figure 8: Loop unrolling versus performance. The highest performance is obtained when neither too many nor too few registers are used.

the unrolling for the programmer. This can be done portably and with modularity if the language being programmed exposes loops and their performance as a built-in, available language feature. Programmers now expect compilers to optimize simple loops, and this has become part of the performance model.

The moral is that portable performance can be obtained, but it can require the programmer to understand and use language (or compiler) features relevant to the mapping of software to hardware, rather than having them attempt the mapping themselves. The programmer talks to a portable mapping facility (in this case, the compiler), expressing the structure of the software to it rather than asking about the hardware and adapting the software.

Parallel Memory Hierarchy model

This section reviews the Parallel Memory Hierarchy model. This is a low-level model, with no notion of threads or objects; however, this model makes it possible to reason closer to the hardware while still allowing the hardware to be abstracted to enable portable high-performance code. This ability to express locality at many levels at once is missing from the Sather distributed extension presented in the following section.

Abstracting the hierarchy

In the PMH model [3] a parallel computer is modeled as a tree of hardware nodes. Each child is connected to its parent by a unique channel. Interior nodes hold data and leaf nodes can perform computation. Data in a node is partitioned into blocks, the unit of transfer on the channel connecting a node to its parent. All the channels can be active at the same time, although two channels cannot simultaneously move the same block.

Each node has four parameters: block size, block count (how many blocks fit in the module), child count, and transfer time (how many cycles it takes to transfer a block between the module and its parent). All modules at a given level are expected to have the same parameters. In the PMH model, the cost of communicating a message is always a step function of the message length. The transfer time parameter can be chosen so that the step function approximates a more accurate model such as latency/bandwidth.

Level	Block Size	Block Count	Transfer Cost
Disk	4K	64K	-
Main	512	12K	~1K
TLB	512	128	~0.06
Cache	16	512	~0.9
Registers	1	32	1

Table 2: PMH parameters for RS6000 Model 530 memory Hierarchy

Table 2 shows the parameters associated with a single RS6000 workstation, in which the descriptive PMH tree has no branches. Notice that the TLB can be modeled as a level of the memory hierarchy, although the transfer cost is abnormally low since only addressing information must be moved, not the data itself. This workstation could be connected to others with a network (as in an SP-2), forming a proper tree.

The p-processor PRAM model is a special case of the PMH model with only two levels, a root representing all of memory with p children, each with unity block size and transfer time. PMH differs from other hierarchical models such as H-PRAM by allowing data to be stored in the interior nodes of the tree. This allows the same structure used to model communication to model the memory hierarchy.

However, PMH is not limited to describing coherent machines. Figure 9 shows two PMH trees associated with two common parallel machine organizations. The tree on the left describes a system in which bandwidth to disk is greater than network bandwidth. In such a system, parallel processing will be limited by the bottleneck of the network. On the right is a system in which the network bandwidth is greater than the bandwidth to disk. The root of the tree represents the combined storage capacity of all the disks, hiding the differences between a workstation's local disk and the other workstations' disks. PMH implicitly assumes full association and complete control over the replacement strategy. Generally, memory in PMH may be manipulated explicitly; there is no distinction between addressable local storage and cache storage.

Making the memory hierarchy explicit can be useful for characterizing serial programs as well. Another model, UMH (Uniform Memory Hierarchy), is a variant of PMH that assumes fixed ratios of block size and transfer time between parents and children, and can be used for asymptotic analysis of communication induced by the memory system for both serial and parallel algorithms [4].

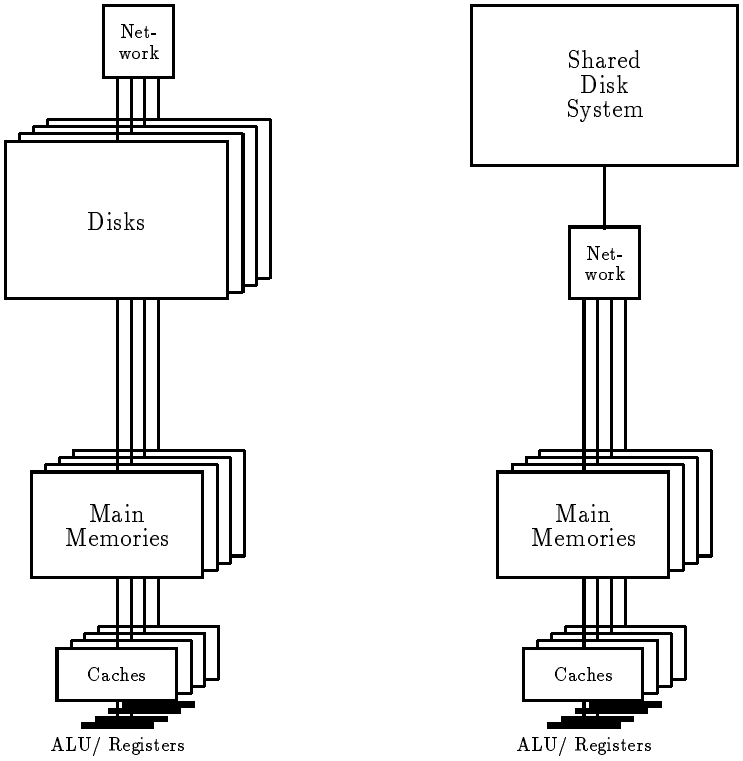


Figure 9: Two trees describing common parallel platforms (from [3]). Dimensions of each block are drawn to be propotional to the log of the block size and count.

Having more than two levels has several consequences. The UMH model focuses attention on the intermediate levels of storage, raising questions about the proper way to subdivide problems. It is also more natural for asymptotic analysis since a single machine can handle problems of all size.

Carter [24] presents hierarchical tiling, a methodology for using PMH to recursively map scientific computations to multiple levels of the memory system. Many efficient hierarchical algorithms for solving numerical problems have recently emerged [26].

A hierarchical model is not appropriate for all systems, and possibly least accurate in describing mesh architectures in which the latency is proportional to the number of hops. The PMH must model a two dimensional mesh as a tree in order to reflect the $O(\sqrt{P})$ bisection bandwidth out of a square of P processors. There will be processors that are adjacent in the mesh but far separated in the tree, so a conservative PMH model may not be able to model systolic array algorithms well. PMH was developed for supercomputing applications, and like CICO, assumes that it is reasonable to describe data as blocks. There is also no provision for threads or non-array data.

Space-limited procedures

Alpern proposes an ambitious development methodology for high-performance portable programming called *space-limited procedures* [6], in which the call graph for generic programs for the PMH model directly reflects the PMH tree structure. Multiple versions of a procedure may be called on, depending on machine and problem parameters. Procedure arguments may further use a *stream* representation. Normally, procedure arguments reside in the module for the duration of the procedure invocation, but for streamed arguments the data is only moved in blocks which can be discarded or moved up to the parent and other blocks moved down.

This is a matrix multiply written using space-limited procedures, in which an $M \times L$ matrix A and an $L \times N$ matrix B and multiplied to obtain an $M \times N$ matrix C:

```
matrix multiply:0 (C[0:M-1, 0:N-1]; read A[0:M-1, 0:L-1], B[0:L-1, 0:N-1])
    assert L<1 or M<1 or N<1
    priority highest
    return

matrix multiply:1 (C[0:M-1, 0:N-1]; read A[0:M-1, 0:L-1], B[0:L-1, 0:N-1])
    assert L=1 and M=1 and N=1
    C[0,0] := C[0,0] + A[0,0]*B[0,0]
    return

matrix multiply:2 (C[0:M-1, 0:N-1]; read A[0:M-1, 0:L-1], B[0:L-1, 0:N-1])
    assert L>1 or M>1 or N>1
    let l = L/2, m = M/2, n = N/2
    cobegin
        matrix multiply (C[0:m-1, 0:n-1], A[0:m-1, 0:l-1], B[0:l-1, 0:n-1]);
        matrix multiply (C[0:m-1, 0:n-1], A[0:m-l, 0:L-1], B[l:L-1, 0:n-1])
    ||
        matrix multiply (C[m:M-1, 0:n-1], A[m:M-1, 0:l-1], B[0:l-1, 0:n-1]);
        matrix multiply (C[m:M-1, 0:n-1], A[m:M-1, l:L-1], B[l:L-1, 0:n-1])
    ||
        matrix multiply (C[0:m-1, n:N-1], A[0:m-1, 0:l-1], B[0:l-1, n:N-1]);
        matrix multiply (C[0:m-1, n:N-1], A[0:m-1, 0:L-1], B[l:L-1, n:N-1])
    ||
        matrix multiply (C[m:M-1, n:N-1], A[m:M-1, 0:l-1], B[0:l-1, n:N-1]);
        matrix multiply (C[m:M-1, n:N-1], A[m:M-1, l:L-1], B[l:L-1, n:N-1])
    coend
    return
```

Each version has a guard assert statements which define its eligibility to be executed. The first two versions of the procedure handle the base cases when the matrices are degenerate or 1×1. The third handles the general case by breaking up each matrix into quadrants and recursively solving the eight matrix multiplication subproblems.

There's nothing particularly interesting about the above procedures, but the next two versions show additional language features that are unique to space-limited procedures.

```
matrix multiply:3 (C[0:M-1, 0:N-1]; read A[0:M-1, 0:L-1], B[0:L-1, 0:N-1])
    assert L>1 or M>1 or N>1
    assert M*N+M*L+L*N <= #this.capacity
    assert Mstride*Nstride+Mstride*Lstride+Lstride*Nstride <= #child.capacity
    hint Mstride ~ Nstride; Lstride ~ 2*Mstride; Lstride ~ 2*Nstride
    priority medium high
    for m from 0 to N-1 by Mstride in parallel
        for n from 0 to N-1 by Nstride in parallel
            for l from 0 to L-1 by Lstride in pipeline
                let mm = min(M,m+Mstride), nn = min(N,n+Nstride), ll = min(L,l+Lstride)
                matrix multiply (C[m:mm-1,n:nn-1], A[m:mm-1,l:ll-1], B[l:ll-1,n:nn-1])
    return
```

This version breaks up highly rectangular problems into more balanced subproblems. It uses the tuning parameters #this.capacity and #child.capacity, which are specific to the hardware the program executes on, which Mstride, Lstride and Lstride are free parameters. The hints express desired relationships between the parameters, but the compilation/execution process may determine them dynamically.

The last version uses stream data; instead of requiring space for the entire arguments for the duration of the procedure, only space for their footprints is needed. When one footprint is no longer needed, it is discarded, or in the case of a result, moved up to the parent and another footprint's worth of data is brought down. In this example, the #registers parameter guarantees that a small block of the original C matrix resides in registers, while a horizontal swath of A and a vertical swath of B are accessed as streams.

```
matrix multiply:4 (C[0:M-1, 0:N-1]; read A[0:M-1, 0:L-1], B[0:L-1, 0:N-1])
    assert L>1 or M>1 or N>1
    assert M*N+M+N <= #registers
    priority very high
    hint L large; M ~ N; M, N as big as possible
    stream A, B with footprints a[0:M-1], b[0:N-1]
    for l from 0 to L-1 in pipeline
        a = A[*,l], b = B[l,*]
        for m from 0 to M-1 in parallel
            for n from 0 to N-1 in parallel
                C[m,n] := C[m,n] + a[m]*b[n]
    return
```

Space-limited procedures allow different versions of software to be executed depending on the abilities of the executing hardware. However, it should be clear from these examples that space-limited procedures are very low level; it is difficult to conceive of their use for maintainable general purpose code. The zones model presented in the next chapter tries to remedy this by allowing expression of locality at the much higher object and thread level.

Sather 1.1 distributed extension

Sather is an object-oriented language that supports parallel threads of execution as well as a distributed extension. The Sather 1.1 specification presents a core language definition and a set of language extensions. The core language is always implemented, while the extensions have hardware or software requirements that make them inappropriate for all platforms. Altogether there are five language extensions: C and Fortran interfaces, threaded parallelism, synchronization, and distribution. An overview of Sather is provided in the appendix on page 99, with the threaded, synchronization, and distribution extensions specified in detail. The distributed extension is also summarized here.

The Sather 1.1 distributed extension is much higher level than the PMH model presented in the previous section; with a single language mechanism, it provides ways to explicitly assign threads and objects to hardware entities. However, it is only able to express locality at the highest level, in terms of clusters. Locality of threads in time cannot be expressed to enable locality-conscious scheduling, nor can locality at finer granularity such as placement on pages and cache lines.

Language features

The memory performance model of distributed Sather has two levels. The basic unit of location in distributed Sather is the *cluster*. The programmer may assume that reading or writing memory on the same cluster is significantly faster than on a remote cluster. A cluster corresponds to an efficient group in the memory hierarchy, and may have more than one processor. For example, on a network of workstations a cluster would correspond to one workstation, although that workstation may have multiple processors sharing a common bus. This model is appropriate for any machine for which local cached access is significantly faster than general access.

At any time a thread has an associated *cluster id* (an INT), its *locus of control*. Until modified explicitly, the locus of thread remains the same throughout the thread's execution. When execution begins, the main routine is at cluster zero. The locus of control of a child thread is the same as the locus of its parent at the time of the fork.

The locus of a thread may be explicitly moved for the duration of the evaluation of a method call by using the binary '@' operator. An expression following the '@' must evaluate to an INT, which specifies the cluster id of the locus of control the thread will be at while it evaluates the preceding method. When iterator calls are on the left side, each iterator evaluation may be placed differently on successive iterations. The '@' notation may also be used to explicitly place forked body threads of fork and parloop statements.

All reference objects have a unique associated cluster id, the object's *location*. When a reference object is created by a thread, its location will be the same as the locus of control when the new expression was executed. A reference object is *near* to a thread if its current location is the same as the thread's locus of control, otherwise it is *far*.

Expression	Type	Description
here	INT	The cluster id of the locus of control of the thread.
where(*expression*)	INT	The location of the argument. If the argument is void or an immutable type, it returns 'here'.
near(*expression*)	BOOL	true if the argument is on the same cluster as the executing thread. If the argument is void or an immutable type, it returns false.
far(*expression*)	BOOL	true if the argument is not on the same cluster as the executing thread. If the argument is void or an immutable type, it returns false.
clusters	INT	Number of clusters. Although a constant, may not be available at compile time.
clusters!	INT	Iterator which returns all cluster ids in order, 0 through clusters-1.

Table 3: Built-in expressions for location in the Sather 1.1 distributed extension.

There are several built-in expressions for location given in table 3. In addition to these built-in expressions, the *with-near statement* asserts that particular reference objects must remain temporarily near. The statement specifies local variables that are asserted to be near while execution remains in the statement body. When checking is on, all assignments to the variables may be tested to make sure this remains true; when checking is off, the compiler may elide code that might otherwise be generated to handle the far case.

Examples

This code creates a object and then inserts it into a table, taking care that the insertion code runs at the same cluster as the table. (The '#' is typed object creation.)

```
table.insert(#FOO) @ where(table);
```

To make sure the object is at the same cluster as the table, one could write

```
loc: := where(table);
table.insert(#FOO @ loc) @ loc;
```

or, equivalently but in parallel:

```
fork @ where(table);
    table.insert(#FOO)
end
```

This code recursively copies only that portion of a binary tree which is near. Notice that 'near' returns false if its argument is void.

```
near_copy:NODE is
    if near(self) then return
        #NODE(lchild.near_copy,
              rchild.near_copy)
    else return self
    end
end
```

The distributed extension to Sather is an example of an annotative model in which threads and objects are explicitly placed on clusters of processors. Explicitly placing objects and threads does not affect the semantics of the original code, but it is also possible to deliberately change the original flow of control (ie. using with-near). It is useful as a reference point in language design because many other distributed languages have a similar approach. This design is a precursor to and will be contrasted with the zoned extension which is developed in the following chapter. The zones model can be viewed as a hybrid of the Sather distributed extension and the lower-level, higher performance PMH model.

Zones

The previous chapters reviewed some of the problems posed by portable, modular high-performance systems. **Memory Systems and Locality (page 14)** reviewed the organization of memory systems and software techniques that try to make the best use of memory systems without requiring programmers to change their ways. The problems that locality creates are too severe for these techniques to handle alone. **Performance Models (page 32)** reviewed alternate proposed ways to allow programmers to express or reason about locality. None of these was found to support simultaneously modular, portable, and high-performance general-purpose code.

The remaining chapters describe a solution to the deficiencies analyzed up to this point. This chapter introduces the zone performance model and details how zones are expressed in the Sather language. The zoned extension provides a level of symbolic indirection between software entities and the hardware they are ultimately mapped to. In this approach the programmer meets the system half-way, coding with special locality annotations that the compiler and runtime use to make placement decisions. With zones, both hardware entities (contexts, processors, clusters, networks) and software entities (objects, threads) are abstracted as trees. The way software zones are mapped to hardware in one implementation is described along with performance results.

The introductory chapter (page 8) explained why it is hard to make high performance systems also portable: high performance demands attention to the details of hardware such as timing, while portability requires abstracting precisely those details away. One way to resolve this dilemma is to create modular systems, where hardware specific code is encapsulated behind portable abstractions. Often a compromise must be made between abstractions at a very high level that favor portability over performance, and abstractions at a lower level that hide most but not all details of hardware. For example, Sather without the distributed extension (page 47) hides almost all details of hardware, while the distributed extension lets the division of the system into clusters show through to the programmer.

The zones abstraction can also show more or less of the hardware details. This chapter presents the simpler of two zone performance models. At the highest level of abstraction, it hides even such coarse hardware detail as gross division into clusters of processors. The following chapter, **Extending Zones (page 81)**, describes how pure zones might be extended to allow more details of the hardware to be seen by the highest performance applications.

PURE ZONES

It is now common to have multiple communication fabrics with widely different properties within a single platform. For example, the workstation this document was typed on has: (1) multiple on-chip register buses, (2) an intra-module bus to cache, (3) a shared coherent bus between multiple processors, (4) a Myrinet point-to-point network between a small cluster of workstations, and (5) switched ethernet for LAN and internet. This is not an advanced or expensive system. Many systems have more levels still.

Each of these levels can differ by an order of magnitude or more in sustainable latency or bandwidth, so it is appropriate to make it possible to distinguish the levels explicitly. Nevertheless, portability requires that the details be hidden. Previous approaches have tried hard to compromise by finding a way to express the details of hardware to the compiler and running program so that software entities can be adapted to match the hardware organization. Because hardware is so complicated, with each level of communication having different performance properties, trying to make the system portable by explicitly exposing all these levels would be theoretically possible but daunting to build and intimidating, at the least, to program. Most importantly, no such portable interface exists, with PMH (page 50) perhaps being the best attempt to date.

		Structural abstraction:	
		Single level	Hierarchical
Communication model:	Explicit	**LogP** (and many others)	**PMH**
	Annotative	**Sather 1.1**	**Zones**

Table 4: Motivating zones. PMH adequately addresses the performance implications of the memory hierarchy, while the Sather 1.1 distributed extension adequately addresses usability.

Table 4 illustrates the problem in terms of the taxonomy of performance models given in the previous chapter. PMH is an explicit model, handling hardware details at multiple levels well at the expense of usability, while the Sather 1.1 distributed extension is annotative, not presenting many hardware details. Zones aspire to the advantages of both.

The pure zone performance model is *nonparametric*: uninformed about the relative costs of hardware operations. This enables portability, but how is it possible to obtain performance this way? Portable performance is achieved in the same way as for registers in loops: by creating language features available to the programmer that are mapped by some platform specific facility onto hardware.

We want to extend this successful idea to the general memory system, but there are additional hurdles. Registers have specific properties that make them unlike other levels of the memory hierarchy; they cannot be addressed and are typically tiny in number, which makes them unsuitable for storing general data structures. An approach that will work for the general memory hierarchy must work for data structures as well. Another property of registers is that they are not shared between threads, so although multiple threads could be ignored in the consideration of registers, they must be considered when modelling other levels.

The software entities being automatically mapped to hardware in the case of registers are local variables, and the extra information passed to the mapping facility is the data dependencies - for example, loop transformations may exploit the independence of values across loop iterations. In the more general case, the software entities to be mapped are objects and threads. It is harder to determine what additional information should be passed by the programmer to enable mapping. The information needs to be rich enough to allow the alignment, clustering, zoning and reordering transformations that reduce misses (page 27). At the same time, this information needs to be sufficiently abstract and high level that modular systems can be designed and maintained.

The choice made here is staightforward and high level: objects and threads are described using a tree, where position in the tree represents the programmers' intuition of locality. Instead of writing code capable of adapting to the organization of hardware, programmers are given a way to describe the organization of their software. This moves the burden of mapping software to hardware from the programmer to the system, where it can be done in any necessarily system specific way. Instead of describing exact data dependencies, the tree representation makes it possible to model approximately a hierarchy of software modules.

The pure zone model

Zones are now described in the abstract, without referring to language features beyond objects and threads. While the zone model could be incorporated into any language capable of expressing objects and threads, the following section then describes exactly how zones appear in the object-oriented language Sather.

Definition

Zones may be summarized in a few sentences:

- A zone represents a set of objects and threads that it is said to own or contain. Zones are themselves objects, so zones may contain other zones.
- Zones form a tree, in which the parent represents the union of its children's objects and threads. Objects and threads in a zone are also considered to be in the zone's parent as well as any ancestor zones. However, zones don't contain themselves.
- Each thread, object and zone has exactly one parent zone, except for a distinguished global zone at the root that has no parent. Hence, the global zone contains all threads and objects other than itself.
- The cost of memory accesses is determined by the smallest zone that encloses both the accessing thread and the object accessed. The amortised time spent for an access is some system specific, monotonically increasing function of the total number and size of threads and objects in that zone.
- Parents are always larger than any of their children: in addition to the child's threads and objects, they also contain the zone object of the child. This implies that if a thread and the memory it accesses can both be arranged to be in a child, performance will be better than if one or both were in the parent or a different child.

The first three points merely define a tree; the last two points are the important ones. They make it possible to reason about performance based on the structure of software alone. Relative costs for different software structures are not available to the programmer, except in terms of inequalities: reducing the number of threads or the size of memory being accessed can't slow things down, and will probably speed things up on most systems. However, the cost is relative to the smallest enclosing zone, not the total number of threads and size of objects in the program. Hence, the programmer is encouraged to place zones around groups of threads and objects that are likely to be accessed together.

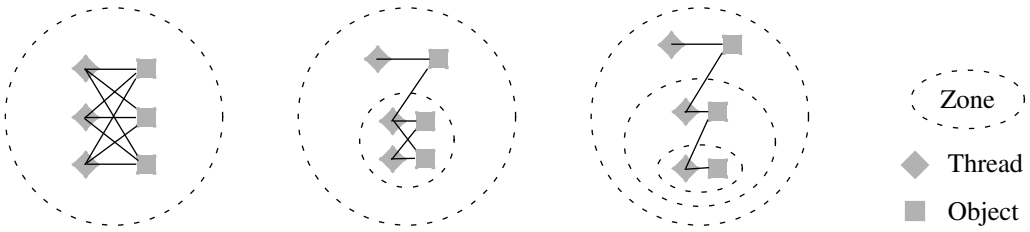


Figure 10: Three zone structures, each expressing different expectations about locality.

Motivation

Figure 10 shows three possible zone structures and common patterns of reference that might suggest their use. The structure on the left has all threads and objects are in the mandatory global zone, but there is no finer structure. This indicates that the programmer

doesn't know of a useful way to group the threads and objects, perhaps expecting random or uniform access. In the middle, the threads and objects at bottom are grouped away from the objects and threads at the global level. This is the typical pattern of use for an encapsulated data structure; the two lower threads and objects form their own zone, and the locality properties of the data structure are not affected by the existence global thread or data structures.

The structure on the right goes further, having the data structure itself recursively express structure. This is how the zone model addesses modularity. In languages like the Sather 1.1 distributed model which allow only a single level of hardware division, the two data structures could not be composed without the locality annotations interfering. With the zone model, they merely compose alongside the objects and threads.

A monotonic model is easy to justify in cache coherent systems. Increasing the number of words being accessed generally increases the miss rate due to capacity or conflict misses. Although there may be many levels of cache, each of them will tend to slow down monotonically as the number of accessed words increases, as the three latency figures in figure 4 (page 18) demonstate. Similarly, misses due to communication and false sharing, as well as overhead due to context switching and synchronization, tend to increase with the number of threads. The effects of bus saturation and false sharing seen in figure 5 (page 22) are evidently monotonic. Increasing the number of threads beyond the number of processors requires sharing processors between threads over time, again incurring some monotonically increasing overhead.

The monotonic model is a generalization of the more common threshold model used on systems with explicit messaging, in which each memory access is either local or remote. In effect, the monotonic timing function for distributed systems on a uniform network is a simple threshold function. If the zone can be contained within a single node, it will be predictably fast. If it cannot because it uses too much memory or there is not enough available parallelism at that node, it will be slower by a fixed latency.

The 'cost' of memory access is purposefully defined without resort to definitions of latency or bandwidth. On a system that hides communication latency with multiple processing contexts, it might seem that monotonicity is violated because overall costs appear to decrease as more threads are added. However, the cost per thread does not decrease; the overall benefit that appears is due to increased parallelism.

One weakness of the zone model is that it relies on the notion of the size of threads and objects, which for most languages cannot be precisely or portably defined. Different implementations may have different word sizes, layout strategies, heap fragmentation behavior and stack allocation, all of which may affect the exact definition of 'size' for a system. However, most systems should have sufficiently similar overheads to allow reasonable portability, given that they only need to remain monotonically related.

Zones in Sather

We now present the second of two distributed extensions to Sather. The first was seen in the section Case Studies (page 40), and presented a flat model of hardware: each cluster of processors was simply identified by an integer. Here, an alternative extension replaces clusters with zones.

Zones are first-class objects that may be manipulated in the usual ways. Zones need not be restricted to concurrent object-oriented programming, but it is convenient to present them that way. Here zones will be defined using Sather, in which ZONE is a primitive class. There are built-in Sather expressions for zones as well as methods provided in the ZONE class.

Built-in expressions

At any time a thread has a *zone of execution*. The zone of execution of main when execution begins is global, that is, all hardware available to the program.

A thread can temporarily change its zone of execution through the '@' operator, and may retrieve its zone with 'here'. '@' evaluates the expression on its left side with the zone of execution set to the zone returned by the right side. After the expression is evaluated the thread continues with the original zone of execution. ('@' may also be used with the fork and parloop statements to change the zone of execution of the body.)

In this code compute is invoked where a resides. (Unlike some COOLs, Sather does not enforce an owner-computes rule.)

```
a.compute @ where(a)
```

All objects have a *zone of residence*. Unlike threads' zones of execution, the zone of residence of a given object can never change. The zone of residence of an object is the same zone as the zone of execution of the creating thread at the time the object was created. The zone of residence of an object can be queried with the built-in 'where(x)' expression. The built-in expressions used with zones of execution and residence are summarized in table 5.

x @ y	Evaluate x with temporary zone of execution y
here:ZONE	The zone of execution of the executing thread
where(x):ZONE	The zone of residence of the object. x

Table 5: Built-in ZONE expressions.

Because Sather threads are not objects, the phrases 'zone of execution of x' and 'zone of residence of x' can be shortened to 'zone of x' without ambiguity. Languages with first-class thread objects would need to resolve the meaning of zone of residence and zone of execution for thread objects.

Zone methods

Zones are objects, and may be constructed like other objects with the create method and the associated '#' operator. The zone of residence of a zone object is the zone of execution of the creating thread at the time of creation. When a thread is created, it may be wise to create a new zone to be its zone of execution: the new zone is a child of the zone of execution of the parent thread. Such a policy of always creating a new zone would attempt to exploit the tendency of threads to operate on independent data. For example, the runtime may interpret the new zone as a directive to avoid placing objects created by the thread where they may cause false sharing with objects from other threads.

Some of the methods in the ZONE class are summarized in table 6. If *y* is the zone of residence of another zone *x*, then *x is within y*; being within is transitive. Alternately, *y* is the *parent of x* and *x* is a *child of y*. The zone method 'x.within(y)' allows this relation to be queried at runtime.

create:ZONE	Create a new zone, which is inside 'here'.
within(ZONE):BOOL	Test if self inside argument zone.

Table 6: Methods in the built-in ZONE class

Examples

Divide-and-conquer is a common computing strategy in which a large problem is broken into smaller subproblems that can each be solved without communication with the others. Such a program has an easy mapping to zones. Because each subproblem accesses only its own objects (and possibly some global state), it makes sense to wrap each in its own zone.

```
solve(arg:FOO) is
    parloop s ::= arg.subproblems!;
    do @ #ZONE;   -- Solve in a child zone
        solve(s);
    end;
end;
```

Another common recursive strategy is branch-and-bound, in which there is communication between subproblems so they do not replicate each other's work. Communication is typically restricted to nodes and their parents, which manage the bounds of their children. In this case, zones still express the locality of communication, which is abstracted as occuring within the zone of each parent.

Zones can also be applied to computations that are not tree-structured by covering the computational structure with a tree. For example, a VLSI simulation might profitably assign zones to regions of a chip and subzones to individual computational cells, abstracting the communication requirements between components into the relative position in a tree.

Zones are least motivated for systolic algorithms and other extremely regular communications for which existing hardware and software techniques suffice. A mesh can be covered by a tree - in fact, many algorithms already divide spatial points using a form of divide-and-conquer to form partitions - but that this is probably not always the best way the communication requirements could be specified.

Zones are intended to complement rather than replace the carefully managed locality in systolic, linear algebra, and FFT algorithms. Existing array-based locality techniques are absorbed into the zone framework by abstraction into nodes of the zone tree; at worst, the tree is just a single zone with the special-purpose code doing all the work. Arrays carry an implicit locality performance model (see page 33), but are simply large objects as far as the zones performance model is concerned.

Comparison with other models

The zone model attempts to combine the object-oriented annotations of the Sather 1.1 extension with the hardware model of PMH. It tries to generalize the notion of locality annotation to be hierarchical and support threads, as well as deal with data structures other than arrays.

In COOL and Emerald, locality is specified by giving threads or objects affinity with other objects. Zones differ by being hierarchical; it isn't necessary to partition the world into a fixed number of processors. This supports compositional locality in libraries. Jade allowed hierarchical specification [81], but dealt with explicit data dependencies rather than affinities.

COOL provides a way to move objects on demand, but this is actually a pragma to move the underlying page. Emerald allows moving objects on demand, requiring the overhead of indirection on every access. Zones may be migrated, but only at the discretion of the system; this design allows potentially higher performance implementation, such as the use of relocating garbage collection to move objects instead of indirection.

Parallel Sather was initially conceived as a language for programming MPPs and NOWs. The previous Sather model [64] was syntactically similar to the zone model, but with a single level of divisions. Instead of recursive zones, the machine was divided into a fixed (runtime constant) number of clusters, each described by an integer. The '@' notation took an integral cluster id instead of a zone.

Two language constructs that made sense for MPPs have been eliminated from the old Sather model. Because hardware was not described recursively, there was no equivalent to *within*, and no way to place threads or objects imprecisely; they were always on a specific cluster. This lead to the relation of *being near*, when two threads or objects are on the same cluster. On machines such as the CM5 where network latency dwarfs local memory access, this is sufficient. The with *x* near...end construct asserted that within the body the identifier *x* must refer to objects near to the executing thread.

Another removed construct, *spread*, exploited the property that MPPs are both flat and well-balanced. On such systems it makes sense to distribute data structures equally among different clusters; the programming effort needed to manually partition the machine into parallel sub-tasks of unequal size can be great and only of benefit for some irregular problems. Spread objects were allocated on a special replicated heap such that a copy of the object resides at the same address on every cluster. This allowed the portion of a distributed data structure local to a cluster to be accessed without referring to a central directory.

The zone view is similar to PMH, but describes software rather than hardware. Unlike PMH, there is no user notion of block size or alignment, with the compiler and runtime expected to allocate data so that it is properly aligned. The following chapter expands the pure zone model to be even closer to PMH: in addition to the software zone tree, a hardware zone tree is available. This tree does not model the memory system as directly as PMH; for example, multiple levels of cache dedicated to a single processor do not receive their own hardware zones. Hardware zones abstract away the notion of memory capacity, but add the notion of parallelism available within a component (thread capacity) to deal with large, flat or dynamic (NOW) platforms.

Expressing locality

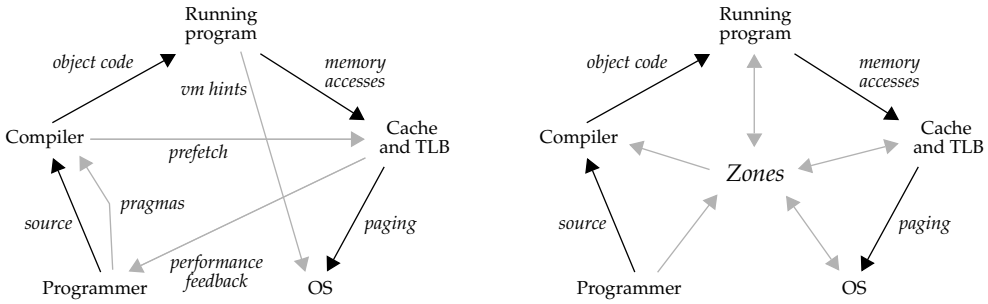


Figure 11: The flow of locality information without and with zones.

Now that zones have been described, we can revisit the flow of locality from the introduction (figure 2, page 12). Zones form a lingua franca with which locality information can be expressed, instead of the conventional piecemeal approach. The same structure that is used by the programmer to communicate to the compiler and runtime - the zone tree - is also used to communicate the structure of hardware to the running code. Because the runtime has enough information to improve performance at multiple levels of the memory hierarchy, zones can be used to structure access to pages; it doesn't require a different set of operating systems calls.

AN IMPLEMENTATION

The previous section introduced the zone model as it is incorporated in the Sather language. Because zones should influence object placement, the implementation of zones requires a special memory manager. This section explains how the zone model has been implemented for Sather. The design goals, organization, and locality optimizations of this general memory manager are presented in depth. This manager is designed to be retargetable to different memory systems and is crafted to respect locality at many levels. The following section Performance results (page 68) describes the applications used to validate the zone performance model, and analyzes the effectiveness of zone annotations towards achieving portable and modular performance.

Special goals of thisimplementation were:

- *Scalable performance* - application performance should be assisted wherever possible by exploiting information about the target memory system. The four optimizations for locality presented on page 27 should be applied; for example, alignment should be used to avoid having objects span cache lines. Similarly, fragmentation, poor locality introduced by garbage collection, and other effects negatively impacting locality should be minimized.
- *Retargetability* - the memory manager should be designed to work with serial, shared memory, and distributed memory systems, and be parameterized by the characteristics of the memory systems. To continue the example, differences in cache line size might result in different placement policies when trying to avoid spanning cache lines.
- *Compatibility* - The memory manager should be compatible with C compilation and existing thread packages. There are unique constraints posed by compilation through uncooperative C compilers, such as difficulty in precisely identifying pointers that are held in registers. For these reasons no attempt was made to explore either relocating objects in memory or custom thread scheduling able to exploit zones.

The general organization of the memory manager is now described along with justifications of how it attempts to meet these goals.

Organization

This section describes the overall organization of memory management, including the data structures used, methods of allocation and garbage collection, and how concurrency is obtained on multithreaded systems. The algorithms used are a synthesis of many approaches; an excellent overview of memory management is [102].

Allocation

Allocation is divided into *regimes*, each of which handles requests for different sizes of memory corresponding to different levels of the memory hierarchy. When one regime is not able to service a request, it is passed up to be handled by a regime that handles larger spans of memory. The regimes are summarized in figure 12.

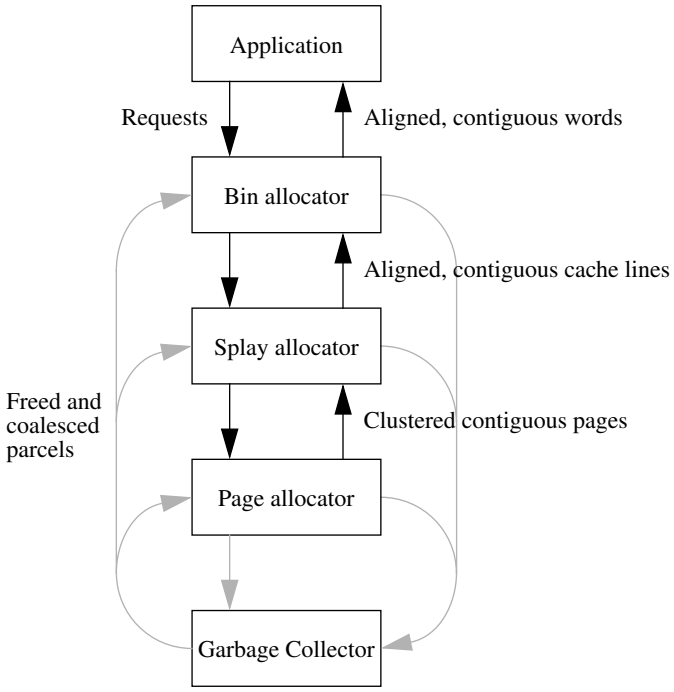


Figure 12: Regimes of allocation. Requests are passed from smaller regimes to larger ones until they can be satisfied.

The many parameters affecting memory management (such as cache line size, page size, word size, minimum alignment, and maximum required heap) are specified by constant macro definitions included into the memory management C code. A separate executable must be compiled for each possible set of memory system parameters, but this also encourages as much specialization as possible at compile time.

Initially, the maximum usable heap is *virtually allocated*. (This heap is distinct from and should not be confused with the heap managed through malloc and free and obtained through the Unix brk call.) Initially, pages in the heap are all mapped to a zeroed, write-protected page; on the first write, a page fault handler maps the affected virtual page to new zeroed physical storage. The result is the appearance of a large, zeroed span of memory which only consumes system resources as it is used, page by page.

Each bit of memory in the heap belongs to a *parcel*. A parcel is a contiguous region of memory that may or may not have an object allocated in it. All objects are also parcels. Parcels are zeroed as the last step before being turned into objects.

C imposes a minimum alignment restriction on allocated storage; on many systems, this is 64 bits. Two bits are permanently associated with each possible address at which an object could begin. The *parcel bit* is set to indicate the beginning of a parcel, and the *start bit* is set to indicate that an object occupies the parcel. Rather than storing these bits in the heap; they are stored in separately allocated, contiguous arrays, usually comprising 1/32 of the size of the heap.

- Objects that are smaller than the target cache line size are allocated in *bins*. Each bin is a collection of parcels obtained as a group; the first word of each parcel points to the next, so that the parcels form a linked list. There is an array of bin header pointers that refers to the next waiting parcel of each size. Allocation requires only checking that the header pointer is not null and reseting it to the next parcel pointed to by the first word. To avoid objects spanning cache lines, when object sizes do not evenly divide a cache line the remainder is linked into the bin list of the corresponding size.

  The number of cache lines allocated at once to form a bin is choosen to make bitwise word operations on the parcel bits fast and convenient. For example, the word with correct parcel bits set for each object size is obtained by precomputed table lookup at the time bins are initialized; no bit manipulations are required.
- Objects that are larger than a cache line, and entire bins of smaller objects, are allocated using a best-fit policy implemented using a *splay tree*. Unlike the bin lists, the splay tree nodes are not stored in the unused words at the beginning of the parcels. Instead, splay nodes are allocated using independently managed bins carved out of entire pages.
- When the splay tree is unable to service a request for an object, a new set of pages is allocated for it. Pages are allocated in groups using address-ordered first fit. The array of bin pointers and the splay root pointer together comprise an *allocator*. A separately allocated *page descriptor table* manages which allocator is associated with each page. Each page only holds parcels of a single allocator, but an allocator may manage parcels on many potentially non-contiguous pages.
- Zones form a tree. Each zone has pointers to children, next sibling, and parent. Zones also each have two allocators: one for objects bearing pointers and one for *leaf objects*, which have no pointers. Different allocation primitives are used to distinguish which allocator should be used. As an optimization, each page descriptor entry also has a bit that indicates if that page may contain objects with pointers. Each zone also has a preferred page in the virtual heap that is chosen at random when it is created; pages are allocated preferentially to the zone's allocators by searching starting at this page.
- Under some conditions, garbage collection is initiated rather than granting new pages to a zone. On distributed systems, zones are also associated with clusters; the details of distributed memory management are discussed in the following chapter.

These data structures are depicted in figure 13.

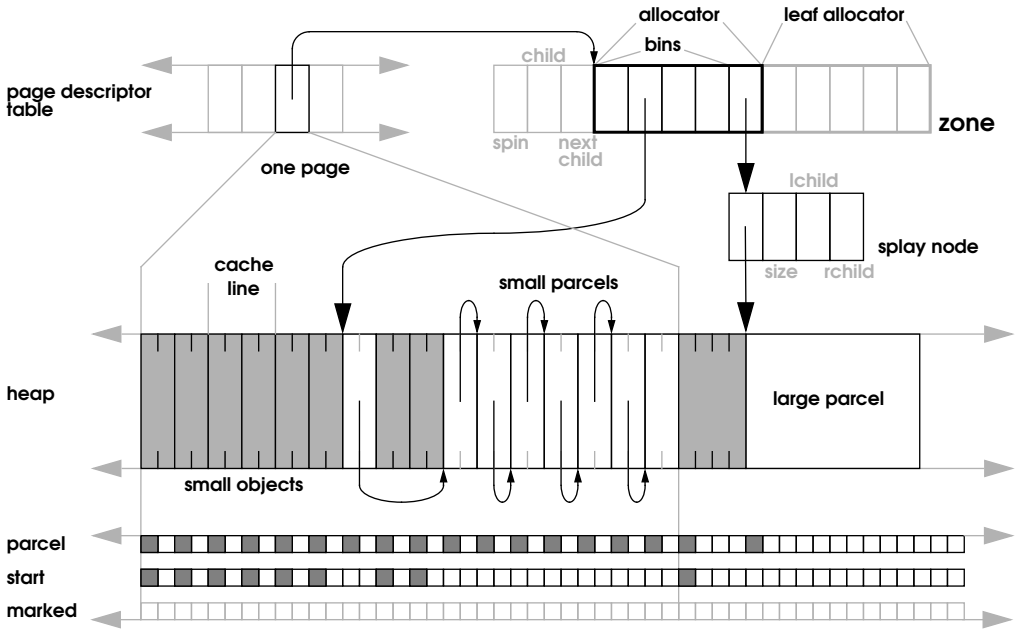


Figure 13: Data structures used for memory management. The zone and splay trees are not shown.

While the algorithms used for memory management are designed for speed, Sather is no more allocator intensive than other imperative languages such as C++. The compiler additionally moves idiomatic allocations to the stack and inserts explicit deallocation for some kinds of garbage known at compile time, so memory management is often not observed to be the computational bottleneck.

Garbage collection

Objects are reclaimed by garbage collection when the number of nonzero pages granted to the heap passes a threshold. On multithreaded systems, collection begins by halting threads other than the one initiating collection. Then all root areas are located and searched for pointers to objects in the heap. For serial systems, this means forcing register contents onto the stack and identifying the stack and data areas. For multithreaded systems, this requires asking the operating system for a general memory map in order to find the stacks of the various threads. These operations of stopping threads and discovering roots are not well supported by most operating systems; this implementation on Solaris requires the use of assumptions about memory layout, assembler routines to save registers, and the use of interfaces intended for debugging.

Collection occurs in two phases. First, all objects reachable from a root are marked. A *mark bit* is associated with each potential object location in the same way as the parcel and start bits. Initially clear, these bits are set when an object is reached during a tight loop that traverses the heap depth-first. Potential pointers are only followed if they point into the

heap, are correctly aligned, point to the start of an object (identified by the start bit), and that object has not yet been marked. The cost of the mark phase is proportional to the combined size of live, pointer-bearing objects in the heap.

The second phase sweeps unreachable objects. First all bin pointers and splay nodes are internally freed by resetting the header and splay root pointers, which occurs in constant time per zone. Then, all information about free parcels is reconstructed. Starting at the beginning of the heap, ranges of pages belonging to individual allocators are considered. Each parcel in the range is considered one at a time. Unmarked objects revert to being free parcels, and are added to the appropriate bin or the splay tree. At the same time, contiguous free space is coalesced. The cost of the sweep phase is proportional to the total size of nonzero heap plus the number of parcels at the beginning of collection. However, the sweep operations are accelerated by fast bitwise logical operations on words and a lookup table to assist in finding the first set bit of a word. Systems with larger word sizes, vector facilities (eg. MMX [53]), or hardware supported population count would benefit even more.

When entire pages are free at the end of a region, they are returned to the pool of free pages, allowing the pages managed by an allocator to shrink. These pages may optionally be returned to the operating system by mapping them to the write-protected zero page again. After sweeping, the threshold for the next collection is set and other threads are restarted, if necessary.

Atomicity and deadlock

On multithreaded systems, accesses to the data structures used by allocation and garbage collection must be protected from interfering with one another. For example, two threads simultaneously attempting to allocate the same size object from a bin could result in each obtaining the same parcel. To prevent this, each zone must be protected by a spinlock which is obtained before the bins or splay nodes of either associated allocator may be accessed. Similarly, the parcel and start bits are protected by the spinlock of the zone holding the allocator which manages them. This serializes accesses to the same zone, but threads which access different zones proceed with only the penalty of uncontested spinlock access.

This simple locking model is upset by garbage collection. A collecting thread has to shut down all other threads in order to find their roots, but in order to make sure internal structures are in a clean state, the collecting thread has to first allow the other threads to complete any pending allocations. However, it isn't sufficient to acquire the spinlocks of all other zones before collecting; more than one thread might attempt to collect at once, which would result in deadlock. This deadlock is avoided by having threads voluntarily give up their spinlock and then reacquire all spinlocks before attempting collection. The spinlock of the root global zone must be acquired before any others; this imposes a deadlock-avoiding serialization to garbage collections.

Design for locality

The memory manager is designed to improve application performance through improved locality. This section examines the design decisions that affect application locality. Also of interest is the locality displayed by the memory management code itself and the effect of that code on application locality.

Fragmentation

Because we may not move objects after allocation, an important concern is the fragmentation of memory. Memory that is wasted (unusable to the application for any reason) may be classified as external fragmentation, internal fragmentation, or overhead. External fragmentation is caused by an allocator not being able to give space to an application because contiguous free space is too small or the policy of the allocator forbids it. Internal fragmentation occurs when the allocator uses sizes that do not correspond to the requested amount. For example, rounding allocation requests for alignment leaves unused memory (eg. when the size is not already a multiple of 64 bits). Overhead is space that is reserved by memory management itself, such as the parcel and start bit tables.

Fragmentation is of great interest because it is correlated with locality; for example, applications which access their allocated objects uniformly would pay miss penalties at every level of the memory hierarchy proportional to the total footprint of memory as determined by fragmentation. Unlike memory system behavior, fragmentation is easy to quantify.

A recent study [73] suggest that conventional wisdom about allocator policy has been skewed by poor experimental method. Two particularly good allocation policies appear to be best-fit, in which the smallest possible free range of memory is used to meet allocation requests, and address-ordered first fit, in which the block of sufficient size closest to the beginning of memory is used. Both have the property that large free regions tend to not be split up when smaller regions are available, and this reduces external fragmentation.

The allocation policy used here is a hybrid of these approaches. Small objects are allocated from bins, which produces little external fragmentation if the bin size is small because objects fit together with no gap. Objects larger than a cache line are allocated best-fit, using the splay tree to efficiently find the next largest available parcel, with ties resolved in a nearly LIFO fashion. Objects large enough to require multiple pages that are not satisfied by the splay tree are allocated with a variant of address-ordered first fit. The variation comes from choosing different pages to be ‘first’ for different zones. While not strictly necessary, this was done to even further discourage mixing together pages from different zones.

Coalescing occurs only at garbage collects, potentially increasing fragmentation when explicit deallocation is used heavily. However, Neely [73] found that deferring coalescing had little effect on fragmentation without garbage collection, which suggests that it is not likely to be a problem.

In this design, internal fragmentation only comes from rounding objects to the minimum alignment required by C, which is unavoidable. However, requiring that small objects not span cache lines is likely to increase external fragmentation; for example, on a system with 32 byte lines and 8 byte alignment, an allocation of 24 bytes leaves an 8 byte parcel remaining. If the application makes no requests for 8 byte allocations, this space will be lost until it can be coalesced into a larger parcel. Sources of additional overhead include the parcel, start and mark bit arrays (3/64 of memory), the page descriptor table (1/2048 with 8KB pages), splay nodes (16 bytes per large parcel), and the stack used during the mark phase, roughly proportional to the depth of objects in the heap.

Conservatism

In order to remain compatible with C, garbage collection is conservative. Unfortunately, this can result in additional wasted space by failing to collect objects that are not actually reachable by the program, but still appear to be reachable because of spurious values in root areas that appear to be pointers. This has not been observed to be a problem in practice, which agrees with our previous experiences with the Boehm-Weiser collector; for this reason, no attempt was made to blacklist areas of the heap spuriously pointed to by root areas.

Various steps were taken to reduce unnecessary conservatism and the impact of necessary conservatism on locality. Careful code generation allows any pointer that does not point to the beginning of an object to be passed over. Code which identifies root areas uses heuristics to minimize the memory scanned. Pointers are distinguished from nonpointers in the heap, for example, by segregating leaf objects. Finally, entire parcels are always zeroed on allocation. Zeroing of the portion occupied by an object is required by Sather semantics, but zeroing any remainder - due to internal fragmentation - helps prevent resurrecting pointers from a parcel's previous incarnation.

Application performance

The principle benefit to application locality is that zones are enabled to the application. Objects in different zones are kept away from each other so that they don't share cache lines or pages, reducing false sharing. Objects in the same zone are also clustered together to minimize capacity and conflict misses.

In addition to zones and the locality benefits of decreased fragmentation and conservatism, the design also encourages application locality by placing deallocated objects at the head of bin free lists and at the root of splay trees. Thus, subsequent allocations retrieve these parcels before others, so reallocation has a LIFO character unless there is an intervening collection. Objects are zeroed only just before granting the space - right before it is likely to be dirtied by the application anyway.

A weakness of this design is that the zone tree is essentially flattened, considering each node of the tree to have equivalent status with respect to placement. While this is a legitimate implementation of the zone performance model, the hierarchy of objects and threads

is not fully exploited. Alternate designs might be able to do so, especially if relocating garbage collection were possible. For example, the present design always keeps objects of different zones on separate pages, but this might lead to fragmentation and conflict misses for an application with many very small zones. Such an application would be better served by clustering that was able to adapt to the sizes of zone subtrees as they change.

No attempt is made to anticipate and avoid application conflict misses. For example, zones smaller than a page will tend to have all their objects at the beginning of the page, which could lead to nonuniform use of cache lines. Bonwick [17] describes a kernel allocator which artifically pads arrays of objects to improve the uniformity and thereby decrease conflict misses. Any attempt to avoid these misses (for example, by allocation starting at a random position in apage) might impose additional fragmentation by dividing large regions or decrease the benefits of alignment within pages, so this was not attempted here.

A size threshold is used to determine when to garbage collect. This threshold is set by heuristics which attempt to balance the locality lost by allowing more pages to be allocated with the performance lost due to collection. At each collection, the time wasted in paging since the last collection is compared to the time spent collecting, and this is used to set the threshold for the next collection. The heuristic attempts to keep the time spent in paging (including paging during collection) approximately equal to the time otherwise spent collecting. Keeping these statistics equal approximates the minimization of their combined overhead; if they are accurately measured and independently monotonic, their sum will be within a factor of two of the theoretical minimum time. This heuristic was found to often be too optimistic about granting memory: applications change allocation and heap access over time, and at the critical point where paging sets in performance degradation can be both nonlinear and catastrophic. For this reason, the heuristic was tempered by hard limits on the amount of heap growth permitted when paging occurs.

Bin allocation code is coded as macros; in the common case when object sizes are known at compile time, constant propagation in the C compiler allows extermely tight, inline code to be generated. For example, which bin to use becomes known at compile time; code can be precisely unrolled to zero the allocated object; and therefore no function call is needed in the common case for allocation. This trick has previously been used for Unix kernel allocation [70]. Other than the use of zones and ensuring that the locality of memory management interferes minimally with the application, there is little else that can be done to improve application locality outright.

Sather guarantees that all assignments must appear atomic to the user. All memory systems have naturally atomic pointer loads and stores, but some larger data types - such as double precision floating point - may or may not be atomic depending on the implementation. To guarantee atomic assignments of other composite types, the Sather compiler must insert expensive spinlock code around the assignments [37]. Atomicity is often determined by the interaction of alignment with the memory system. For example, a double word that does not cross cache line boundariesmay be atomic with respect to other processors because communication always occurs using the contents of entire lines in write-back caches. Since the memory manager aligns all small objects to cache lines already, that suggests that restriction of these objects to the heap could allow the safe elimination of many spinlocks. (This problem is not restricted to Sather: in a concession to x86 semantics, Java only man-

dates atomicity for double precision variables declared to be volatile. As there is no way to declare the elements of double precision arrays to be volatile, hence traditional scientific codes are less robust against race conditions.)

Memory management locality

Splay nodes are isolated from the memory they describe. Splay nodes are only needed for objects larger than a cache line, so the use of a parcel header that holds the splay tree pointers in-line would lead to very poor use of that cache line. Instead, splay nodes are allocated with their own bins on their own pages, which brings greater density to the splay information itself; more than one splay node fits on a cache line, and it is not necessary to access the memory being considered for allocation to allocate it.

Splay trees obtain log time amortized access by modifying the tree at every access to bring the accessed node to the root. This gives them some unique properties among tree data structures. Recently accessed nodes get moved closer to the root, so repeated queries for similar sizes become very fast. Programs frequently allocate similar sized objects together, so this may improve allocator efficiency.

Because splay nodes may be written to on every access, the importance of isolating splay nodes from the data they describe is especially important to avoid costs associated with writing - for example, false sharing and disk writes of dirty pages. A final potential cost of placing splay nodes in free areas is that such areas are likely to be aligned to common offsets from within the cache. For example, object bins are always aligned to a convenient multiple of cache lines. If splay information was frequently located at those offsets, this could cause extra conflict misses, especially on direct-mapped cache hardware.

The placement of start bits in a table distinct from the heap data the bits describe similarly avoids needing to access memory to find out whether it contains a useful parcel or is the start of an object. Memory associated with objects needn't be accessed at all during the mark phase to identify legitimate pointers, and the segregation of leaf objects in the heap allows marking to often proceed merely by checking the page descriptor table first - far less expensive than a potential page fault. Similarly, the sweep phase only accesses lines in which application data may reside to link together binned parcels. The sweep phase accesses the parcel, mark and start bits in a regular, streaming fashion in a single pass from one end of the heap to the other.

Summary

An implementation of the pure zone model has been created for shared memory machines. All aspects of memory management were crafted with an eye towards the locality of the application as well as the locality of memory management itself. Allocation uses different policies depending on the size of memory requested, tuned to the size of cache lines and pages. Policies were chosen to reduce memory wasted due to fragmentation, while the algorithmic implementation of those policies was chosen for efficiency and locality. Internal data structures for memory management are segregated from application

data. Major design constraints include the need to cooperate with C compilation and the use of existing thread packages. These constraints resulted in conservative garbage collection, concurrency in allocation but not collection, and no attempt to schedule threads.

The idea of hierarchically structured memory management is not new. The locality of Unix kernel allocators is better understood [96] than that of general purpose memory management. Dynix [69] used a multitiered allocator, and Sciver et al. [83] partitioned allocations into regions distinct regions based on object type[3]. Hierarchical partitioning of allocation itself, however, appears to be novel.

Performance results

The previous section described how this implementation maps software entities onto target hardware with memory management parameterized by the target memory system. This section now evaluates this implementation of the zone model.

Two microbenchmarks are used to show that the zone model can be relevant to performance. Each takes a simple operation on data local to a thread and explores scalability as more threads are added, with and without zones. Keeping the operation performed by each thread simple makes it possible to interpret performance effects that would be too subtle to observe in larger bodies of code. The first operation examined is linked list reversal, and the second is matrix multiply for various numeric element types.

Two more substantial Sather applications are presented to demonstrate annotation by zones in real code. The first application is the Sather compiler. Because it was written before zones were proposed, this code is an honest data point in evaluating how useful zones can be at decoupling algorithmic structure and locality annotation. The second application is a physical simulation with an efficient algorithm that requires tree-structured computation and therefore suggests the use of zones.

List reversal

Table 7 shows inner-loop code that is contrived to be limited by the memory system. It reverses a linked list by scanning down the cells, reversing one link at a time. Each list has a thousand objects, each of which is eight bytes. Reversing the list requires performing memory reads and writes on each object, and these cannot be optimized away by clever compilation. When multiple threads are operating on multiple lists at the same time, the placement of objects on cache lines will affect performance. Changing the zone annotations used by the code which allocates the data structures should change the observed performance.

3. This paper called allocation regions *zones*, although they were nothing more than contiguous arrays of equally sized objects. Recursive *zones of zones* were considered, but instead the recursion was truncated at the second level by a special zone allocator. The implementation described here might be honestly appraised as making the same compromise: it does not attempt to exploit the structure of the zone tree beyond distinguishing individual nodes.

```
prev:FOO := void;
p ::= lists[i];
loop until!(void(p));
    n ::= p.next;
    p.next := prev;
    prev := p;
    p := n;
end;
lists[i] := prev;
```

```
...
loop:  st %o2,[%o1+4]        -- reverse pointer
       mov %o1,%o2           -- prev := current
       orcc %o0,%g0,%o1      -- current := next
       bne,a loop            -- continue if not void
       ld [%o1+4],%o0        -- branch delay slot
...
```

Table 7: Microbenchmark code designed to be limited by the memory system. Left: Sather code to reverse a linked list; right: resulting sparc instructions for the inner loop.

Three types of zone annotations are timed:

- **Zones** - zones are added to organize the touched objects so that each thread's objects are within a single zone. The zone performance model predicts that this will result in higher performance, because the enclosing zone for each thread contains fewer threads and objects (see definition on page 53). In terms of the memory system, objects used by different threads can be expected to never share cache lines, minimizing false sharing and capacity misses.
- **Random zones** - zones are added which have no meaningful relationship to the expected patterns of memory access; this controls for the effect of having multiple zones.
- **No zones** - with no annotation, all object allocation occurs using the default zone allocator that main is started with. Objects are allocated to the lists round-robin; this order ensures that objects from different lists end up sharing cache lines. On multithreaded systems this is expected to lead to false sharing.

These three policies are used to explore the portability and modularity of the microbenchmark. For multiprocessors, a primary performance concern is scalability. Figure 14 shows the scalability of inner loop memory accesses of the three zone policies on a Sparc 10 with 128MB and four hypersparc processors[4]. Additionally, the timing of purely serial code (including optimizations available only to serial Sather) and the ideal speedup are shown. Each data point represents running the microbenchmark many times to amortize start-up overhead; the maximum speed over at least 25 trials was taken to remove transient effects caused by other processes. The dips at 5 and 9 threads reflect the nonpreemptive scheduling of threads under Solaris, leaving processors idle.

It can be seen that zone annotations are an effective way to achieve scalability in this microbenchmark. Unzoned allocation causes the application to become swamped by false sharing misses as soon as two or more processors are executing, together performing worse than simply running serial code. The extremely flat speed for serial execution coincides for all three allocation strategies; the cache easily holds an entire list while it is repeatedly reversed, even when poorly allocated. The slight performance increase after 10 threads is probably a result of the list reversals growing out of sync; false sharing drops when lists are less likely to be accessing the same position in the lists at the same time.

4. Elapsed times produced with the cs 1.1 compiler with -O_fast and gcc 2.7.2 with -O2.

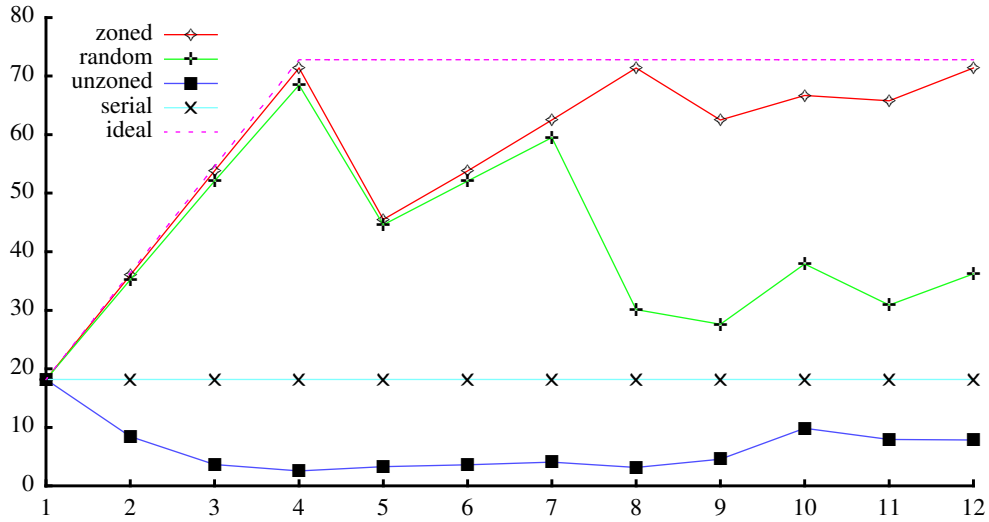


Figure 14: Performance on a Sparcstation 10 with four hypersparc processors. Vertical axis is combined list reversals per millisecond; horizontal is number of threads (one list per thread).

The zone performance model predicts that random zoning will at worst be somewhat inferior to having no zone annotations at all: the enclosing zone of each thread is still the default, global zone, which is now bigger (because it contains the zone objects). Here, random zones eliminate most false sharing misses, but decrease the cache efficiency by not clustering successive list nodes as closely in memory. The probability that list items share cache lines decreases with more threads, bottoming out at only one object per accessed cache line. Because objects are eight bytes and cache lines are 32, this amplifies the working set by a factor of four. As a result, random zone allocation holds up until there are eight threads; at that point, the combined working set grows beyond 256KB, the point where L2 cache and TLB misses set in on the hypersparc[5]. In contrast, zoned lists stay nearly optimal because the working set stays much smaller than the cache.

Figure 15 shows the effect of executing the same code without modification on two other platforms with different memory systems: ultrasparc and pentium pro uniprocessors. These serial platforms do not suffer from false sharing, but they are still affected by capacity misses created by not having the data as densely organized in memory. The ultrasparc's L1 cache is twice the size of the pentium pro, so degradation for unzoned allocation sets in at three zones instead of two.

Interestingly, unzoned allocation does slightly better than zoned allocation for a single list on the ultrasparc. No zone annotations means that all objects, including Sather runtime structures, are allocated from the unique default zone. The slight performance increase might be due to reduced conflict misses by allowing these tangential data structures to be clustered with the list data. While having no zone annotations eliminates all cache line re-

5. Details of the three memory systems benchmarked in this section may be found in table 15 on page 111, and are plotted for size and stride vs. latency in figure 4, page 18.

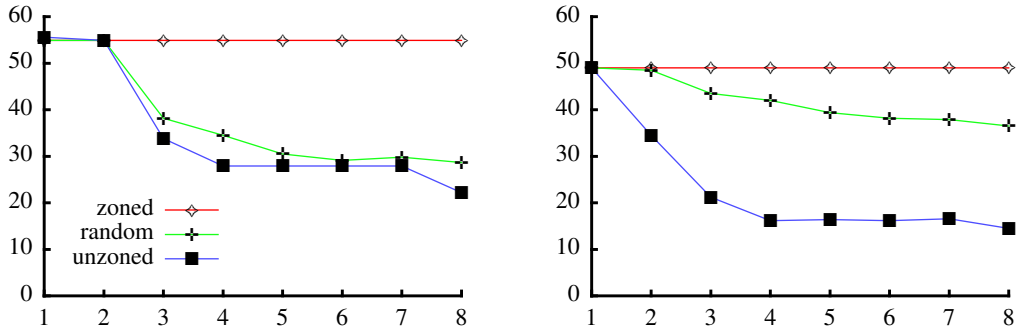


Figure 15: Performance on serial platforms. Axes same as figure 14. Left: ultrasparc. Right: pentium pro.

use for four or more threads, the random zoning does admit some reuse and so does marginally better. Most importantly, zoned allocation consistently retains approximately peak performance for all numbers on both processors.

Portable performance has been demonstrated by demonstrating that zone allocations can maintain efficiency on three different system with varying numbers of processors, cache sizes, and compilation models (serial and threaded). Modular performance has been demonstrated by performance scaling over a range of data structure sizes and degrees of parallelism.

Matrix multiply

The list reversal microbenchmark was clearly contrived to cause memory system problems; for example, the order of list cell allocations was controlled to invoke false sharing. This might not represent the performance of real applications. We now examine the composition of unmodified library data structure code with and without zones.

The Sather libraries provide a variety of mathematical types, including matrix and vector classes that can be parameterized by the type of element they contain. For this microbenchmark, various library classes were composed without modification to create matrices with the following element types:

- FLT, the IEEE single precision floating point type. A matrix of this type is stored densely with the same layout as Fortran. FLT is a base type in Sather and its properties are given special consideration by the compiler.
- CPX, an immutable complex type, composed of two FLT fields for the real and imaginary components. Because this class is labeled immutable (see page 103), the compiler also knows how to densely allocate the matrix in the same layout used by Fortran.

- OBCPX, a type identical to the library type CPX but without explicit immutability. In this representation, a new object must be allocated on the heap for each operation such as addition and multiplication. The matrix is thus represented as a contiguous array of pointers.
- RAT, rational numbers represented by an arbitrarily large integral numerator and denominator. Each RAT is immutable, so there are two pointers per matrix element. In addition to the contiguous matrix storage, each element also requires two heap objects, one for each of the infinite-precision INTI objects representing the numerator and denominator.
- RATCPX is a complex number in which each field is a RAT. Because both CPX and RAT are immutable, the representation this generates is again a contiguous matrix, with each element requiring four pointers.

Both FLT and CPX have performance models which are *implicit*. Sather programmers know that FLT is a base type so its performance model is defined by the language. The language also guarantees that arrays of immutable objects will be allocated contiguously. The other element types do not have implicit performance models. Zones are designed to allow performance modeling without exclusive reliance on implicit models or explicit placement. At the same time, they should coexist with existing methods for obtaining locality, and so must not degrade performance in the FLT and CPX cases.

Because Sather guarantees atomic assignment, the naive use of an immutable class in a parallel system introduces expensive implicit synchronization in the form of spinlocks (page 66). There are three ways to work around this problem. The traditional solution is to collapse the data structures by manually inlining all occurrences of the immutable class. This substantially hinders code reuse. A second solution is to give the compiler special license to eliminate the implicit synchronization - in essence, promising the compiler that there are no race conditions in the code. A third way is to avoid immutable objects and use ordinary heap allocation, the natural implementation in object-oriented languages such as Java that have no equivalent notion of immutability or aggregate values. The second and third alternatives were taken here.

Figure 16 shows the relative performance of these element types and the effect of adding zones on performance on a Sparc 10 with four hypersparc processors and 128MB. Each data point was obtained a large number of matrix multiplications many times and recording the fastest; times include overhead for allocation but not garbage collection. The inner loop is literally

```
loop repeat.times!;
    m := m * m;
end;
```

where m is initialized to the 10×10 identity matrix for that type, to avoid potential numerical performance issues such as exceptional underflow. When zones are enabled, there is one zone per thread.

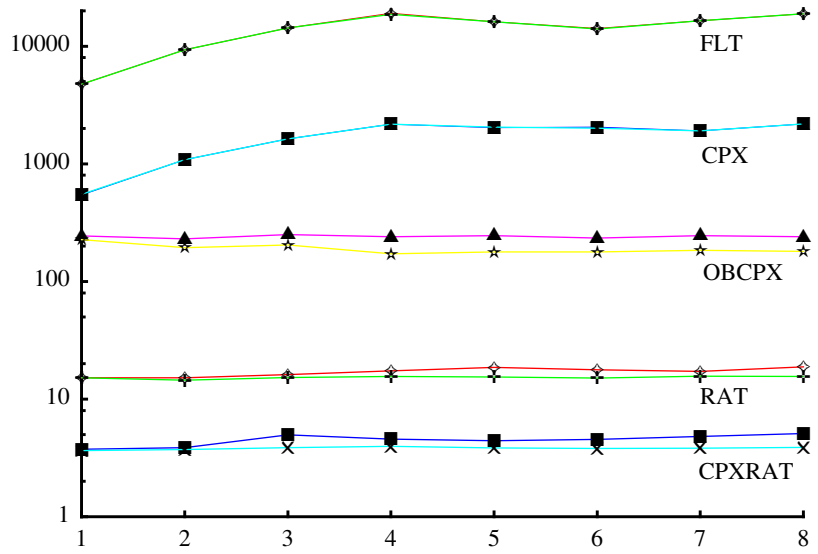


Figure 16: Number of threads (horizontal) vs. matrix multiplications per millisecond (vertical, log scale). For each element type, the upper line is performance with zones, the lower without zones.

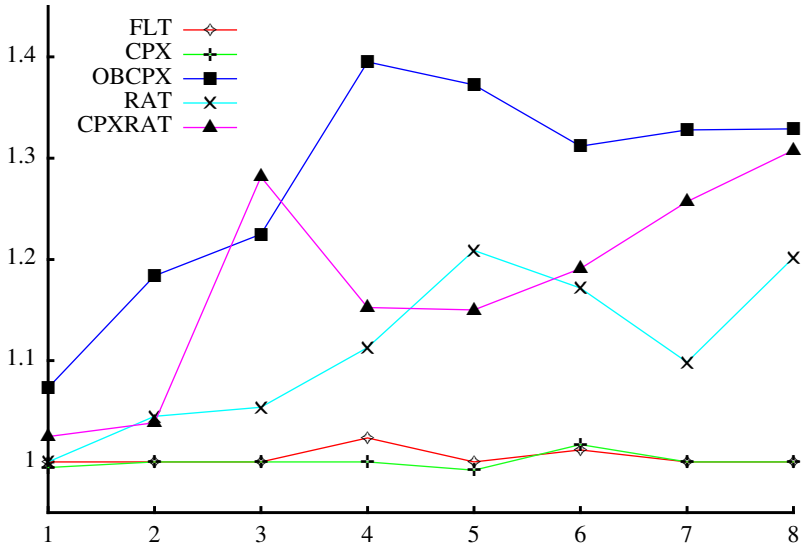


Figure 17: Number of threads (horizontal) vs. speedup for adding zones for each element type.

Each element type loses roughly an order of magnitude in performance over the previous. The types with an implicit model scale nearly perfectly with the number of threads; the contiguous representation eliminates indirection and makes efficient use of cache. In contrast, the more general data structures do not scale as well. These structures require heavy object allocation; for example, each intermediate value of a row-column product requires

generating an object which is referenced only once, when it is consumed. Under these conditions, the system bus becomes a bottleneck and the scalable behavior of allocation is essential. Multiple zones help increase allocation concurrency as well as improve cache behavior.

Figure 17 shows the ratios between the speeds of the zoned and unzoned implementations. Note that these speedups might be better viewed as negative slowdowns; they were obtained by preventing loss of performance through contention, but actual scalability through parallelism remains poor on this platform.

Sather compiler

The Sather compiler is the largest application considered here, about 42k lines of code, not including the 40k lines of the Sather standard library. This compiler has been widely distributed and been in production use for several years. (Details of the design can be found in the section 'Implementation Overview' in the appendix on page 104.)

Compiling is an application domain offering many opportunities for zoning to improve locality. Consider scanning and parsing, in which a number of files are converted into abstract syntax trees. Each node represents a grammatical language construct. The stream of characters in each file is scanned into a stream of tokens, which is parsed into the tree; a symbol table is constructed at the same time.

This line is from the parser in the loop that parses all files.

```
tcd:AS_CLASS_DEF:=parser.source_file
```

If it is desired to keep all the abstract syntax nodes that came from a single file clustered in a single zone, a new zone can be created for the routine to execute within; this will cluster together every object the routine creates.

```
tcd:AS_CLASS_DEF:=
                parser.source_file @ #ZONE
```

A similar annotation can be used to cluster by module (groups of files).

Note that when the routine source_file returns, the zone of execution of the calling thread also returns to what it was before the call. The Sather compiler uses a recursive descent parser with a routine that corresponds to each language construct. If it were desired to cluster tree nodes by class or method as well as by file, this can be done with additional annotations; the generated software zone tree would then follow the syntactic structure. This would be an example of the modularity of zones. If other phases of compilation access the syntax tree with locality that coincides with this choice of zoning, performance may improve (for example, by reducing conflict and capacity misses.) However, there is a limit to which the granularity of syntax can be profitably zoned because of overhead, particularly in this implementation which consumes at least one page per zone.

No parallelism was involved in this example. If threads were forked to parse each file concurrently with a new zone for each, the clustering of objects from the same file would be automatic. Multiple threads, however, would also require making sure the shared symbol table data structure remains safe and efficient when accessed by multiple threads - for example, through mutual exclusion. While adding zone annotations never violates correctness, adding parallelism to a complex originally serial code could and was not attempted.

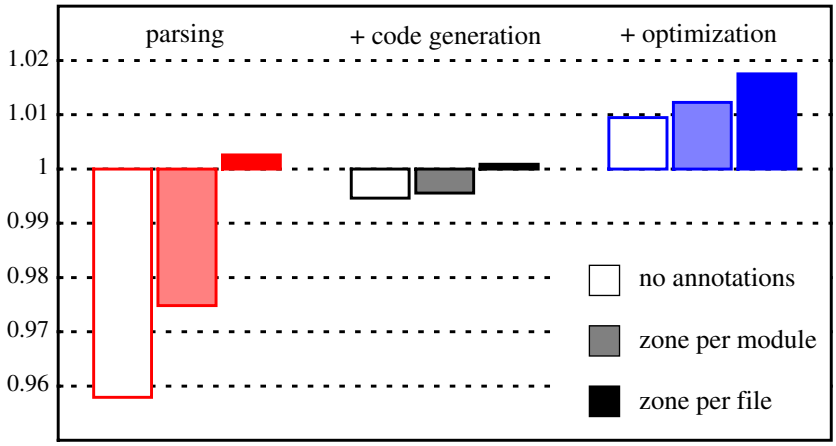


Figure 18: Speedup of zone annotations, normalized to the Boehm-Weiser collector.

Figure 18 compares the overall performance of the benchmark applications using the zone memory manager against the performance of the Boehm-Weiser collector. Times are the minimum elapsed time on at least one hundred runs on an unloaded Ultra I with 128MB. To enable this comparison, each application was compiled to execute serially with no overhead for parallelism and with full optimization. Three annotation variations are shown: no zones; an annotation creating a single zone per module; and an annotation creating a single zone per file. There are also three variants of compilation: stopping after the parsing phase; stopping after C code was generated by ordinary, non-optimizing code generation; and a full compilation including `-O_fast` optimization. In each case compilation was halted before the generated C code was written to disk, so the times include only input I/O overhead.

Parsing is a small portion of compilation. The total elapsed time for parsing was around four seconds, so measurements were easily dominated by start up overhead. Parsing is primarily an exercise in I/O and allocation, with little opportunity to exploit well-placed data. Unannotated compilation fared poorly with respect to the Boehm-Weiser system. However, longer running compiles saw greater benefit to the locality-conscious design of memory management design.

In each case, having a zone per file brought a speedup, but this system does suffer from a fragile dependency on good annotations. Experiments with zoning on a finer granularity brought poor results, presumably because of the high overhead of individual zones. This

suggests that the design would benefit from a more adaptive policy towards the use of pages, perhaps by allowing sharing of pages until sufficient allocation within the zone warrants the higher order of partitioning.

When a new identifier is encountered, it must be inserted into the shared symbol table. Without farther annotations, each identifier would likely be clustered with the syntax nodes of first file that referenced it. Depending on the input, this might be either good or bad; an alternative strategy would be to build a symbol table which dynamically places identifiers in the zones of the classes that reference them most often.

Adaptive mesh refinement

Adaptive mesh refinement is used to efficiently simulate continuous physical phenomena; for example, the motion of waves. The behavior of fluids can be described by partial differential equations. These equations may be approximated by constructing systems of discrete equations representing values at points on a grid. Computers model the phenomena by numerically solving the discrete system.

Unfortunately, fine grid resolution is often necessary to faithfully model the physics of interest. Fine grids have many variables, so the system of discrete equations is generally too big to solve using exact techniques. Instead, iterative numerical methods are used that compute successively more accurate approximations to the solution. Naive application of these techniques may still take too long to converge, requiring a number of iterations far more than linear in the number of grid points.

Fortunately, many physical phenomena display variations in scale that can be exploited. Multilevel methods [18] use coarser grid resolutions to speed convergence on the finest grid. Furthermore, there are often portions of a simulation for which a coarser grid will suffice. For example, modeling a shock wave may require fine resolution around the discontinuity of the shock, while the areas away from the shock are quiescent and may be modeled with tolerable accuracy with only coarser resolution. Adaptive mesh refinement (AMR) locally refines the mesh only where it is needed, allowing finer resolution of the problem for the computation expended.

A typical AMR code partitions the physical domain into patches. Any point in the grid is covered by at least one patch, and patches can nest to bring arbitrary resolution where it is needed. It is convenient for patch resolutions to have twice their parent's resolution. This naturally leads to a tree representation, in which depth in the tree corresponds to finer resolution and children are patches nested and aligned with the coarser parent patch.

Figure 19 shows snapshots from a simulation by the author that uses AMR to model four ripples propagating in a shallow viscous fluid. Also shown are the patches used at each stage of the simulation.

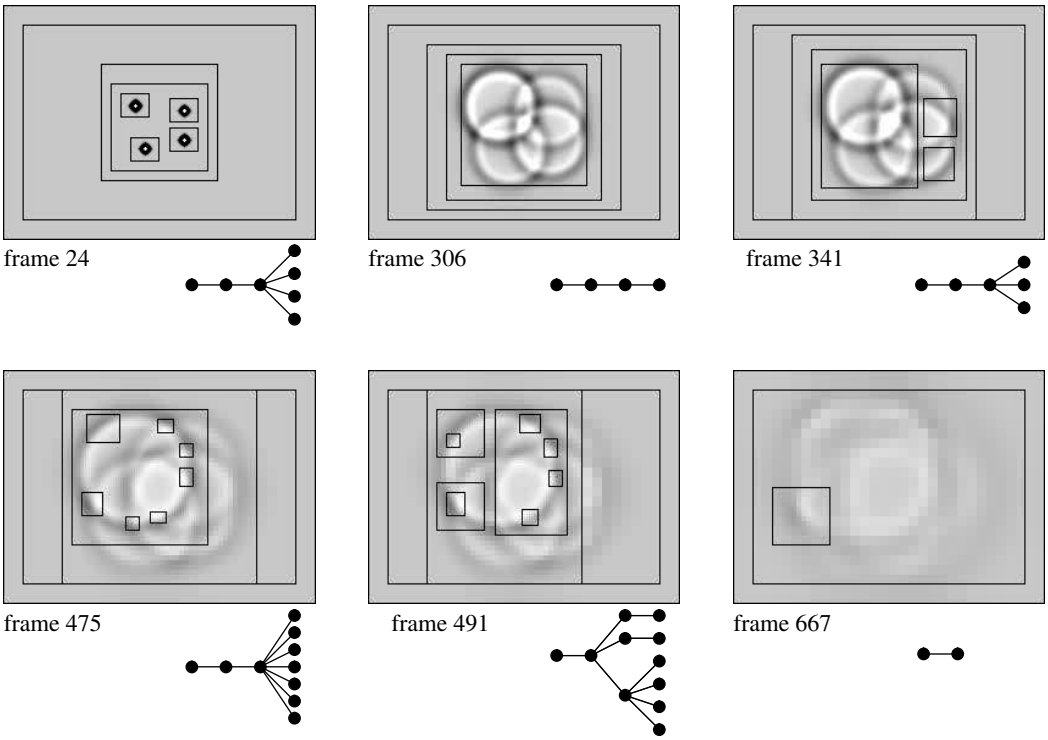


Figure 19: Frames from the droplet simulation and the corresponding adaptive patch trees.

In this implementation, each iteration advances the entire system by a constant time step. At each time step, each grid point is dependent on its immediate neighbors within a patch. Grid points on the edge of a patch must also match boundary conditions imposed by the neighboring child or parent patch. Patches are refined by comparing results on a coarse grid to the results obtained on a finer grid; when the error per unit area exceeds a threshold, a new, finer patch is created. Patches may shrink when this error drops again. For details of the specific algorithm chosen for representing patches, see [66].

Each patch is represented by a data structure with two arrays of grid cells. One array represents the previous time step's values while the next time step is computed. Each array element is a composite structure composed of the floating point variables modeling hori-

zontal and vertical velocity and fluid height. This simulation is parameterized; to change the physics being simulated, it is only necessary to plug in a different cell class. Each array is ringed with 'ghost cells' that represent values outside of the patch.

The entire system of patches is evolved over time in synchronized iterations. Patches may compute the evolution of their cells in parallel to their siblings and parent, but must synchronize to compute boundary conditions and to consider whether patches represent sufficient resolution to meet accuracy goals. The tree structure of parallel computation is expressed by this recursive routine using a par statement.

```
iterate is
    par
        project_from_children;
        compute_ghost_cells;
        loop
            c::=children.elt!;
            fork c.iterate @ where(c) end;
        end;
        evolve;
    end;
    consider_repatching;
    swap_arrays;
end
```

The computation required to evolve a patch at each time step requires access only to the memory representing its grid points and is independent of communication with other patches. Communication and synchronization between patches is only necessary between a patch and its parent and children; therefore, communication always follows the edges of the patch tree. This suggests associating each patch with a zone to contain its arrays and children.

The natural representation of cells are as immutable objects. Such objects can be efficiently manipulated in registers or on the stack, and arrays of immutable objects are contiguous in memory (page 103). The implementation with immutable cells is already well structured for locality without zones, because each array of cells is contiguous.The only expected benefit in this implementation is the clustering of the two arrays representing the previous and current time step. Figure 20 illustrates speedup on a Sparc 10 with four hypersparc processors and 128MB, with maximum concurrency throttled by a global semaphore. The upper pair of lines shows that adding zones has very little effect on performance.

In contrast, heap-allocated elements are only contiguous in memory to the extent that consecutive allocation requests happen to perform contiguous allocation. Heap allocation also consumes an obligatory object header word. The lower pair of lines in figure 20 illustrate speedup relative to the same control simulation as the upper pair, the immutable cell implementation running serially with the Boehm collector. The penalty relative to this control simulation for single threads is 37.8% without zones.

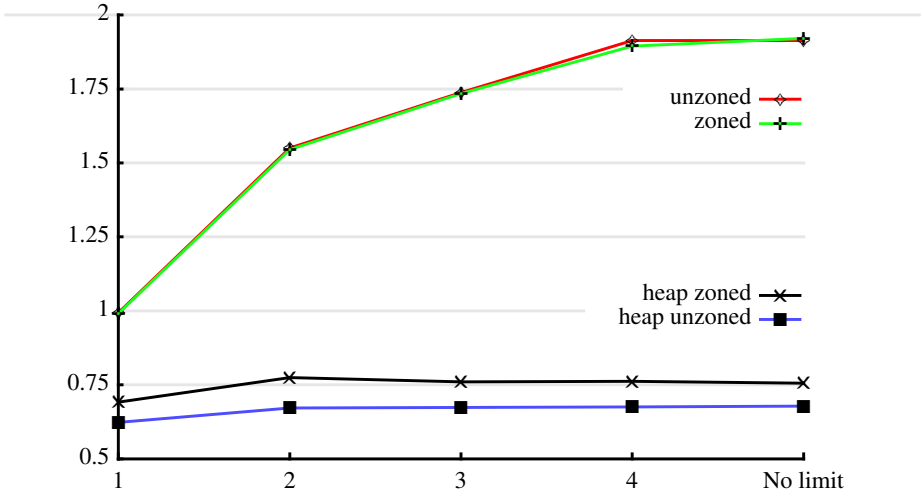


Figure 20: Speedup (vertical) of AMR code relative to serial code with the Boehm collector for varying maximum concurrency (horizontal) on a Sparc 10 with four hypersparc processors.

While this implementation of zones does guarantee that cells within a single patch are clustered in memory and is expected to provide improved performance during the interior evolution of a patch, it is not able to exploit the more general tree structure that determines communication between patches. The implementation with heap allocated cells does benefit somewhat from zones. With unlimited threads, the speedup with zones relative to no zones is 12.1%. On a uniprocessor Ultra I with 128MB the equivalent speedup was 13.2%, so this effect is clearly not due to increased concurrency of allocation. However, this benchmark remained unable to scale on this platform as threads were added.

Assessment

One of the goals for this implementation was compatibility with existing usage of compilers and thread packages. This had an overwhelming influence on the design. In particular, relocating objects and zone-aware thread scheduling were ruled out, although these could reasonably be expected to play a crucial role in the effective implementation of zones. As a result, only a marginal implementation of the zone model was achieved.

The list reversal microbenchmark conclusively exhibited pathological memory system behavior, but it was specifically contrived to do so. The other applications demonstrated speedup with zones; however, the link between this speedup and zoned allocation was not ironclad. It is possible that these benefits came from misunderstood performance effects. For example, enhanced concurrency of memory management could perhaps explain some of the benefit seen on multiprocessors.

More general goals of the zone model are high performance and modularity. This implementation obeys the letter but not the spirit of the zone performance model by only attempting to cluster objects with respect to individual zones. While the performance of the implementation was at least comparable to the Boehm memory manager, modular performance was not demonstrated. Only one benchmark (AMR) used a true zone tree structure, but this application already displayed admirable locality within patches. The implementation was not able to exploit the structured communication between patches. The resulting speedup was uninspiring.

Conclusive demonstration of the practical benefits of the zone model requires more than attempted here. Practical validation of the model would include an implementation of zones that more closely implements the performance model, demonstration that performance benefits are due to memory system effects, and applications composed of sufficient layers to explore the performance effects of composing independently zoned code.

Extending Zones

The previous chapter, **Zones (page 50)**, described the pure zone model, an implementation, and performance results. While entirely portable and modular, this model may not be sufficient to obtain the highest level of performance on machines with severe penalties for poor locality. The lack of information about the hardware was partially compensated for by giving the programmers a rich way of expressing their intention towards software entities, but more information about the structure of hardware could still be very useful. For example, networks of workstations, in which conventional computers are used as a single coordinated computing resource, are often limited by overhead for using the network. Obtaining high performance on these platforms requires that the structure of memory access reflect the structure of hardware. This, in turn, requires that the structure of hardware be visible to executing code. This chapter proposes an extension to the pure zone model that would make this information available.

The next chapter, **Conclusion (page 94)**, summarizes the other chapters and finishes with possibilities for further research.

HARDWARE ZONES

There are many ways to describe hardware to the programmer. The approach taken here exposes different levels of hardware by abstracting the hardware as a tree. Like PMH, the leaves of this tree represent hardware units capable of executing threads, such as processors, or thread contexts within a processor. Interior nodes represent collections of hardware. They are the combination of all hardware represented by descendent nodes, and have two or more *divisions*, each of which forms a subtree. The root represents the entire machine on which the program executes. The programmer knows that communication within a node is faster than communication with nodes above it on the tree, but nothing about the relative performance between levels. This exactly mirrors the performance model for software zones.

Leaf nodes have a integral *capacity*, which is a suggestion of the number of threads that are available in parallel, indicating how many threads should be created to maximize utilization. The capacity of an interior node is the sum of the capacities of its children. A hard-

ware tree consisting of a single leaf node at the root describes a system only in that it specifies a recommended number of threads; it does not commit to any hardware structure beyond that. Capacity, like the structure of the hardware zone tree, is a runtime constant. (Users of the pure zone model are also able to discover the overall thread capacity of the machine, but it is not a notion tied directly to zones.)

A tree structure need not be imposed on the description of hardware. For example, if an aggressive memory system is present, latency effects may not dominate execution time (cf. Tera, Cray T3E). On networks of workstations, which workstations contribute to the large single system image may vary during a program's execution, so it is not accurate to describe the hardware with a fixed tree. Such systems may be best described by a single node.

Like PMH, it would further be possible to have even more information about block size, latency, bandwidth, transfer times, heterogeneity, and so forth. Many of those detailed parameters do not fit particularly well into a high-level view of objects and threads, so they are not pursued here.

Zone hardware abstraction examples

The previous chapter anticipated that many future systems will consist of loosely coupled processor clusters. The zone hardware abstraction is flexible enough to represent these as well as most existing computing platforms, demonstrated by these examples.

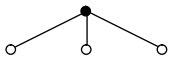
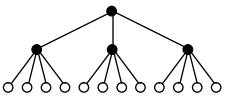
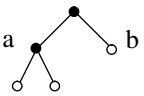
A	B	C	D	E
				

Table 8: Five trees associated with the example platforms

1. The most common computing platform, a single processor workstation, is represented as a single node with no descendents (see table 8, **A**). It is not useful to divide the processor into smaller units, so this solitary root node has no descendents and a capacity of one. Unlike PMH, caches do not receive their own nodes.
2. A network of three workstations, each with four processors sharing a bus. If the shared bus of each workstation is fast enough that explicit application support is not required to avoid contention, then **B** could be used; each leaf represents a workstation and has a capacity of four. If the effects of sharing a bus may be a problem, the tree can be extended to give a leaf to each processor as in **C**. In this case, the leaves have unit capacity but the interior nodes still have a capacity of four.

   There are other possibilities. **E** could be used with twelve leaves each with unit capacity if the bus local to each processor is much faster than the shared bus but the network is not significantly slower, which might be appropriate for SCI connected clusters. **A** would be used if the number of workstations could vary dynamically, with a single leaf of non-unit capacity.

3. An wide-area network of redundant workstation clusters. In this case (**D**), there are two cooperating institutions, **a** and **b**, connected by the internet. Both institutions run fault-tolerant single system image operating systems such as Solaris MC [58]. Institution **a** has two such system images connected by a high bandwidth fiber network. The actual number of available nodes in any system image may vary as users come and go, so no attempt is made to abstract that hardware entity as a fixed number of divisions. The capacity of each leaf is set when execution begins, based on system history, other loads, or other heuristics.
4. A large MPP, such as a T3E or CM5. Supercomputers typically have a very expensive communication fabric that can obviate the need for software attention to locality (**A**) or result in a flattened tree (**E**).

The hardware tree in Sather

Just as zones are associated with software entities, in this extension zone objects are also associated with each hardware entity. Zones created by the programmer are *software zones*, and zones describing hardware are *hardware zones*. The tree of software zones is extended by the *hardware zone tree*, the runtime interface describing the hardware to the program. The hardware zone tree is created by the system before a program beings and must not change during execution (nodes may not appear or disappear).

However, the hardware zone tree does not have to be a compile-time constant, and may change from run to run. For example, on a network of workstations in which nodes may be added on-the-fly, it is permissible for new executions to see the new nodes as long as currently running programs do not. It also might be more productive for systems to distribute concurrent jobs into separate hardware units instead of time-sharing an entire machine. The zone tree would allow libraries on such a system to customize their behavior for the hardware they are assigned for that particular run. Of course, compilation optimized for a fixed target (i.e. by compile-time constant propagation) is also possible.

In the proposed extension, zones describing hardware and software form a single tree. The hardware zone tree follows the within relation; contexts are within processors which are within clusters which are within the whole machine. Taken together, the hardware and software zones form a unified tree, with global at the root. Only those zone objects belonging to the hardware tree represent a piece of hardware, and these additionally support the methods are summarized in table 9:

global is a builtin expression that returns the root of the hardware zone tree. divisions returns the number of divisions of the hardware zone, and division(*n*) may be used to select one. As a convenience, the iterator divisions! is equivalent to division(divisions.times!). divisions returns zero if the zone is a leaf of the hardware zone tree. Table 10 illustrates the use of these methods for the sample hardware zone trees presented in table 8.

global:ZONE	The root of the hardware zone tree. Special expression construct.
divisions:INT	Number of divisions, if a hardware zone. If a leaf zone, returns zero.
division(INT):ZONE	Return zone of n^{th} division.
divisions!:ZONE	Yield zone for each division.
capacity:INT	Thread capacity of hardware zone.
division_of($OB):INT	Division of hardware zone that argument is within; the hardware parent

Table 9: Some built-in methods for examining the hardware zone tree. All but the first are methods of the ZONE class.

Expression	Tree A	Tree C	Tree D
global.divisions	0	3	2
global.division(0).divisions	*illegal*	4	2
global.division(0) .division(0).divisions	*illegal*	0	0
global.capacity	*runtime constant*	12	*runtime constant*
global.division(0).capacity	*illegal*	4	*runtime constant*

Table 10: Expression evaluations for example platforms

Example

This code finds the maximum depth of the hardware zone tree rooted at the current zone of execution. Threads are forked off recursively for each division. The mutual exclusion variable 'm' is used to avoid a race condition when updating the local variable 'res'.

```
depth:INT is
    m ::= #MUTEX;
    res ::= 0;
    parloop div ::= here.divisions! do
        d ::= depth @ div;
        lock m then res := res.max(d) end
    end;
    return res + 1
end;
```

The implementation is responsible for binding each software zone to a hardware zone. If the zone is a leaf of the hardware zone tree, then the placement of the thread or object to a hardware entity (such as a processor) is exactly determined. However, if the zone is an interior node of the hardware zone tree, the runtime may place the object or thread anywhere within that hardware entity. Software zones, such as new zones created by forking

threads, may similarly be placed anywhere within the most restrictive hardware entity they are themselves within. When the operations on hardware zones (table 9) are applied to software zones, they instead operate on the hardware zone to which that zone is bound.

The binding of the software zone tree to the leaves of the hardware tree must be transparent and may be dynamic. For example, the runtime may migrate threads or objects for load balancing, and the movement of data in a cache coherent machine may be viewed as fine-grain migration. When not bound explicitly to a hardware leaf at creation, there is no way to later query which hardware leaf is being used for a particular thread or object; programmers must express their locality requirements through the use of software zones and trust the runtime to place them appropriately.

The division of the hardware zone which encloses a given object is returned by division_of(x). It is an error if x is not within the hardware zone of self.

Placement examples

On platform 2 (three workstations each with four processors) this code would have the following implications:

- A new software zone is created for main.
- A new software zone under the software zone of main is created for the thread t1, and any threads or objects it creates will reside in that zone and thus may be placed at, or migrated to, anywhere in the system.
- Any objects t2 creates will reside in the first workstation's memory and any threads will be executed by one of the first workstation's processors.
- t3 will be executed only on the first processor of the first workstation.
- ob1 may reside anywhere.
- ob2 may reside only on the first workstation.

```
main is -- Zone of execution is global

    first_cluster ::= global.division(0);

    :- t1;
    :- t2 @ first_cluster;
    :- t3 @ first_cluster.division(0);

    ob1 ::= #FOO;
    ob2 ::= #FOO @ first_cluster;

end;
```

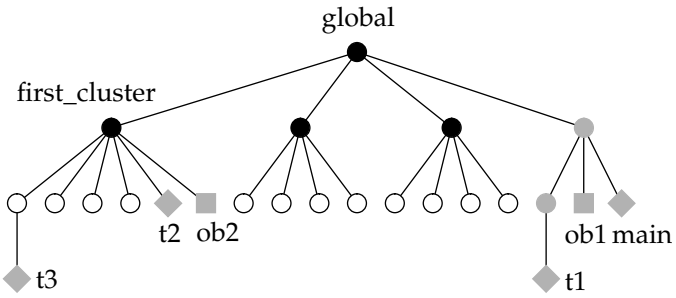


Figure 21: Zone tree created by code example. Black nodes are interior hardware zones, white nodes are leaf hardware zones, and grey nodes are created by the program. Edges show the within relation. Circles are zones, squares are objects and diamonds are threads.

The zone tree created by this program is shown in figure 21. The two grey zones were created implicitly at thread creation. The above code is not portable because it does not

check whether the root zone is divided before asking about its divisions.

This code tests the object 'ob' to see if it is within the zone of execution. If it isn't, a local copy is created.

```
if where(ob).within(here) then
    local_ob := ob;
else
    local_ob := ob.copy;
end;
```

Caching

Memory systems can be broadly categorized into those that cache remote references locally and those that do not. The section Memory Systems and Locality (page 14) surveyed the mechanisms by which caching is implemented by hardware, software, or both. The important difference to the programmer is that caching systems have a very different performance model in which reads become fast at the expense of making writes slower[6]. A tuned caching system allows the memory system resources to scale with the number of readers without imposing a central bottleneck.

Algorithms can have profoundly different scaling behavior depending on how the underlying memory references scale with the number of processors. A portable hardware abstraction needs to reflect this, making it possible for a programmer to exploit caching when it is available without requiring it when it is not. For example, on a network of workstations cache coherency is implemented in hardware within a workstation but must be implemented in software, for a high price, at the network level. The overhead of caching at the network level must not be imposed on programs that do not need it in order to scale.

In this extension, classes that depend on caching behavior are marked with the cached keyword. This tells the compiler and runtime that the programmer depends on scalable read performance. Classes that are not annotated in this way are not guaranteed to have scalable read performance.

6. The cost of a write may be considered amortized over subsequent reads. Consistency models are orthogonal to the memory system performance semantics in that they only change the timing dependencies of reads and writes to the thread, not their relative costs when they do occur.

Example

Distributed data structures frequently require a directory in which to record the location of the components; such structures are typically written just once at initialization, making them ideal for caching. This class behaves just like an array, but avoids a central bottleneck by requiring the runtime to cache remote accesses if this is not already provided by hardware.

```
cached class DIR{T} is
    include ARRAY{T};
end
```

EXAMPLE: DISTRIBUTED VECTOR

This section demonstrates the construction of a portable distributed vector class using extended zones. Examples include routines for creation, vector addition, random element access, and iteration. These routines are typical of the coding required for any distributed class which has a fixed structure after creation and supports both parallel synchronous operations and asynchronous random access.

On parallel platforms there are often vector libraries specifically tuned to the hardware. The code below is not intended to supplant machine-specific libraries, which should always be used in place of portable but slower code. An application remains portable if it uses libraries in which vectors are implemented in the best way for that machine. This vector implementation can be seen as a fall-back when special libraries are not available, and a demonstration of coding techniques that are applicable to other data structures for which tuned versions are infeasible. In particular, distributed hash tables can be constructed using very similar code.

In this implementation, the distributed vector structure is determined at creation by examining the hardware zone tree. The internal representation is also a tree; vector operations are implemented by recursively dividing the vector into subranges. Leaves of the tree are

the primitive vector class VEC. A supertype $DVEC over VEC and DIST_VEC allows both leaves and interior nodes to be children of any interior node. The framework looks like this:

```
abstract class $DVEC > VEC is                  -- Supertype
    plus(arg:$DVEC):SAME;                      -- Vectors may be added to each other
    elt!:FLT;                                  -- Vectors may be iterated over
    ...       -- Other methods not shown
end;

cached class DIST_VEC < $DVEC is               -- Subtype
    private include ARRAY{TUP{$DVEC,INT}}      -- Tuples of children and first index
        elt! -> ARRAY_elt!;                    -- Rename array iterator; we'll redefine it
    const minrange:INT:=64;                    -- Minimum tolerated elements in a leaf
    ...       -- Other methods not shown
end;
```

Because this distributed vector class is built on the primitive VEC class, it will automatically take advantage of any special compiler or hardware support for dense vector operations. Zone annotations are applied to the VEC objects without affecting the correctness of their code. Zones may generally be added to existing code to improve performance without affecting correctness, and complement conventional coding for locality such as explicit tiling and prefetching.

Creation

The creation routine is shown in table 11. It builds the distributed vector tree recursively in parallel, following the shape of the hardware zone tree. At each branch in the tree, a decision is made about how may subranges to divide into to meet the following constraints:

1. There must be exactly one DIST_VEC object per interior node.
2. There may never be more threads in a subtree than its capacity.
3. There may be no more than minrange elements per thread.
4. Subranges should balance the vector by capacity of the subtrees.

When subtrees complete their creation, they fill in the tuple array elements in the parent DIST_VEC object.

A major liability of this creation routine is that there is no optimization of placement according to the memory system details. For example, on some systems it would be better to avoid distributing across the upper divisions because more is lost in overhead than gained in parallelism. On other systems, not distributing a vector could leaed to a working set larger than the cache size. The optimal choice depends on details of the memory structure and usage patterns, and sometimes subtle changes in placement can effect dramatic shifts in performance [48]. The zone model extension does not attempt to provide performance portability to this degree, but does not rule out compiler optimizations, machine specific

```
private div_up(a,b:INT):INT is return (a+b-1)/b end;--Integer divide, rounding up.

create(sz:INT):$DVEC is
    -- Compute total number of threads to be created in all leaves
    elts_per_thread::=div_up(sz,capacity).min(minrange);
    threads:=div_up(sz,elts_per_thread);
    children:INT;                                   -- Compute number of children
    if here.divisions=0 then
        children:=threads;                          -- At a leaf of the hardware zone tree
    else
        children:=threads.min(divisions);           -- Not at a leaf
    end;
    if children=1 then return #VEC(sz);             -- If a single child, just make a leaf
    else
        res::=new(children);                        -- Otherwise, make a new DIST_VEC
        parloop                                     -- Create children in parallel
            subsize:INT;                            -- Size of this child
            z:ZONE;                                 -- Zone for this child
            i::=children.times!;
            if here.divisions=0 then                -- At a leaf
                subsize:=elts_per_thread;
                z:=here;                            -- Place within current zone
            else                                    -- Not at a leaf
                t::=here.division(i).capacity.min(threads);-- Distribute by thread capacity
                subsize:=t*elts_per_thread;
                threads:=threads-t;
                z:=here.division(i);                -- Place on child hardware zone
            end;
            start:=sum!(subsize);                   -- Index is sum of prior sibling sizes
        do
            res[i]:=#TUP(#SAME(subsize) @ z, start);
        end;
        return res;
    end;
end;
```

Table 11: Distributed vector creation routine

libraries, or more detailed extensions to the model. For example, hardware zones could be extended with methods to query expected bandwidth, latency, and overhead. Such information could be profitably used here without requiring further language changes.

Vector addition

Table 12 shows distributed vector addition, a bulk parallel operation which can be computed recursively in parallel. The vectors being added together are expected to be aligned; both the vector data in the VEC leaves as well as the interior DIST_VEC nodes are expected to be placed as they would have been by create. A vector tree created at the same zone will always have subranges placed at corresponding zones. This property is exploited by creating the result vector aligned to self without needing the more complicated code in create.

```
plus(arg:$DVEC):SAME
    pre size=arg.size and where(self)=here          -- Require correct alignment
is
    typecase arg when DIST_VEC then                 -- Always true if aligned
        res::=new(size);
        parloop i::=size.times! do                  -- Add children in parallel
            chunk::=[i].t1;
            argchunk::=arg[i].t1;
            assert where(chunk)=where(argchunk);    -- Assert is true if aligned
            res[i]:=#(chunk+argchunk @ where(chunk), [i].t2);
        end;
        return res;
    end;
end;
```

Table 12: Distributed vector addition

The assert statement that preceeds the subrange addition indicates the programmer's intention that the nodes are aligned; only local communication is needed to perform the addition for leaves. Assertions are a kind of checkable comment, verified at runtime if checking is on. The compiler may exploit the assertion and generate code specialized for the known co-locality of chunk and argchunk. The precondition in the routine header may also be used in this way.

This code creates a new vector for the result. High performance computing would more typically add in place, storing over the original value for self. Such an addition method would be very similar but would suggest additional synchronization at the top level to detect or avoid concurrent operations modifying the value of self.

Random access

In Sather, the bracket '[...]' notation is syntactic sugar for a call on the routine aget. Table 13 shows an implementation of aget for the distributed vector. Random element access

```
aget(idx:INT):FLT is
    loop                                        -- Find which child index is in
        i::=0.upto!(size-2);                    --   (could also use binary search).
        while!([i+1].t2<idx);
    end;
    child::=[i].t1;
    startidx::=[i].t2;
    return child[idx-startidx] @ where(child);  -- Pass request to child recursively.
end;
```

Table 13: Distributed vector random access

is different from the previous operations in that it must be very lightweight; it is important that many concurrent accesses do not bottleneck at a central directory, such as the root of the distributed vector tree.

There is expected to be much less data in the interior nodes where all directory information resides than at the leaves. In bulk synchronous operations such as parallel add, the directory nodes are each visited once, as is each leaf. For random element access, however, the interior nodes are visited once for each element below them.

Here it is important that the DIST_VEC class is labeled cached (see page 87). The interior nodes of the vector tree are ideal for caching because once initialized, they are never again modified. The first reference to the top node will stored locally and subsequent accesses will be fast. Nevertheless, this will require extra data replication; although the interior nodes are much smaller than the leaves, the number of interior nodes visited on any element access is proportional to the height of the tree, so for some sufficient depth the memory required to replicate the directory will dwarf the memory used for actual data. Similarly, hardware trees with extreme branchiness (i.e. **E** in table 8) increase the amount of directory data that must be cached. On cache coherent systems, eventually this directory data will cause capacity misses in the local cache; on systems with compiler supported caching there may be a similar loss of locality and diminishing space available for vector data.

At the time of this writing most systems are flat, with at most three levels. Those systems which are very branchy are MPPs which tend to have very aggressive low latency memory networks (T3E, Tera), suggesting a single global node of high capacity rather than a tree. In short, the directory caching problem is expected to be the worst on those machines on which it matters the least, because they do not have deep or branchy hardware trees. MPPs are also more likely to have special tuned libraries. The old language construct spread (page 58) was intended to deal with precisely this problem of replicated directory structure and remains a complementary possibility.

Exploiting iteration locality

Sather supports iterators at a language level, treating them as methods like routines [72]. Table 14 shows a vector iterator customized to take advantage of the locality present in the leaves, copying each leaf vector to the zone of execution before iterating over it. The copy method in VEC is primitive, and can be trusted to use efficient bulk transfers when copying remotely.

The code for vector addition required that the argument be aligned to self. Alternately, the code could have accepted a misaligned structure and made local aligned copies of the leaves explicitly as above.

```
elt!:FLT is         -- Naive implementation
    loop
        c::=ARRAY_elt!;
        loop yield c.elt! end;
    end;
end;

elt!:FLT is         -- Implementation that exploits locality
    loop
        c::=ARRAY_elt!;
        typecase c
        when VEC then
            if ~where(c).within(here) then    -- Is it local?
                c:=c.copy @ here;             -- If not, get a local copy
            end;
            loop yield c.elt! end;            -- Known to be local
        when DIST_VEC then
            loop yield c.elt! end;
        end;
    end;
end;
```

Table 14: Distributed vector iteration

Conclusion

The first three chapters discussed prior work on locality and concluded that there are severe limitations with existing methodologies. The last two chapters introduced the zone model and an extension to address these problems. Now we conclude with a summary of what has been achieved and the shortcomings of this work, which suggest directions for further research.

SUMMARY

The methods used to create high performance systems often directly interfere with methods for programming in the large. In particular, portability and modularity are often directly at odds with the needs of performance.

This thesis proposes a solution to part of this problem: how to deal with memory systems. Caching memory systems maintain the illusion that all memory access is uniformly fast. However, this is a deception. Hardware trends have been increasing the distance to memory so that performance is becoming increasingly dependent on software locality. With existing methods, software locality is in turn dependent on memory system specifics; as a result, performance is neither portable nor compositional.

Perspective on locality

System components - memory systems, applications, compilers and operating systems - are unable to truly cooperate because they are not able to talk to one another. Obtaining good locality requires knowing target memory system characteristics - such as the sizes of blocks, cache sizes, and division into processor clusters. There are no standardized, portable interfaces through which memory system characteristics can be discovered by software. There is also no standard representation of locality characteristics of software that can guide the compiler and runtime. The lack of standard representations forces systems to be constructed of many independent pieces, each attempting to solve part of the locality picture in isolation, or at best with clumsy, piecemeal information.

Specific examples of this failure to communicate include:

- Applications may ask the operating system about the page size, but it is difficult or impossible to discover cache sizes or which pages are currently resident.
- Compilers may insert prefetching or vector instructions to inform the memory system about future accesses, but they must do so blind to the programmer's intended locality relationships for the objects the code manipulates.
- Operating systems can attempt to reschedule threads on processors on which they previously executed to improve temporal locality, but are unable to know if doing so may create false sharing with another currently scheduled thread.

A more expressive representation of locality is required to allow this cooperation between components. To facilitate programming, however, the representation must be usable by programmers in daily practice. This requirement lead to the review of locality performance models and the development of a taxonomy to describe them.

Locality performance models may be categorized as *implicit* or *explicit*. Most performance models are implicit, requiring programmers to understand lower level implementation of the programming system they use. For example, high performance programming with arrays on vector machines requires some understanding of how compilers vectorize. In contrast, explicit performance models simply force the programmer to deal with hardware limitations directly, requiring the use of explicit messaging and matrix and vector libraries. Some explicit models may be further distinguished as *annotative*. Annotative models are a compromise allowing application expressiveness and correct semantics to not be impacted by locality; for instance, compiler directives in comments may suggest vectorization without requiring the structure of code to change.

Zones

The high level, annotative *pure zone model* was proposed to rectify the limitations of existing performance models. Pure software zones describe the locality structure of software by organizing it as a tree. The programmer relies on the compiler and runtime to dynamically map threads and objects onto hardware units such as processor clusters, pages, or cache lines. This indirection allows programmers to reason in the abstract about performance instead of worrying about the complexities of specific hardware.

Pure zones are a rigorously portable abstraction because they do not allow any representation of hardware. Pure zones enable modularity, because the recursive zone tree may be used to describe the composition of software modules. However, the zone tree is also easy to extend to incorporate concrete details about memory system hardware when that is desired. For example, the penalty for communication between processing nodes may be so large that it is desirable to structure computation to reflect the underlying hardware. One possible extension to the pure zone model is the *hardware zone tree*, which describes divisions of hardware using the same tree used for software.

An implementation of the pure zone model was created for the ICSI Sather system. This implementation made informed use of the parameters of the underlying memory system and gave consideration to the locality of memory management. Improved performance was demonstrated for a handful of applications and was most impressive on shared memory systems. This implementation was limited in significant ways by limitations of the implementation such as compilation through C, poor operating system support for threads, difficulty of multiprocessor memory system analysis, and a shortage of large Sather applications for benchmarking. The observed performance, in spite of these limitations, underscores the potential of zones when these limitations are overcome.

Zones are least motivated when existing techniques for obtaining locality suffice - such as for codes that can be expressed in terms of dense linear algebra operations. However, in many programs there is not such a kernel which can be microengineered for locality. For these programs, zones can complement the locality of carefully engineered kernels and provide the modular glue needed to obtain true portable memory system performance.

FUTURE DIRECTIONS

The implementation of zones described for the ICSI Sather system was effective but not particularly aggressive; all benefits were gained only from changing layout in memory, but there are many other possibilities. The section considers directions for further work on zone implementation and performance models.

Migration and scheduling

Compatibility with C compilation is an enormous boon to portability, but severely constrains memory management. Not being able to identify pointers precisely makes it difficult to relocate objects. If objects were relocatable, the zone tree could be used to guide their placement during garbage collection with not much overhead beyond that required by conventional copying collection. For example, objects could be placed with all objects of a single zone together contiguously in a depth-first traversal of the zone tree. This layout would produce locality that would more closely mirror the performance model that the tree suggests than the implementation discussed here.

The counterpart to the placement of objects in memory is the placement of threads in time. Although the zone model addresses both threads and objects, this implementation did not attempt to modify thread scheduling to execute threads in related zones together or to guide their placement on processors or clusters. The principle reason for this omission was the lack of Solaris system facilities for manipulating thread schedules and placement. The present implementation has to operate blind, guessing to get initial object placement right and relying on Solaris for thread scheduling.

The distributed implementation is not yet fully functional. Enabling the migration of objects and threads between nodes would allow load balancing heuristics to make use of the zone tree, potentially allowing both improved locality as well as higher processor utilization. When objects and threads can be moved, it is possible to recover from a poor initial choice of placement as well as adapt to different phases of computation.

A project is underway to replace Solaris threads with a custom facility that will bypass the Solaris thread implementation, making it possible to coschedule threads for locality and efficiency [100]. Another possibility is the integration of the management of Sather high-level locks with zones. For example, a choice between which of two threads to unblock could be made in favor of the temporal locality suggested by the zone tree.

On-line performance feedback

It is difficult to analyze systems as large as an executing multithreaded Sather program. In particular, it is hard to obtain information about what the memory system is doing. Conventional tools for understanding memory system behavior rely on gathering traces and analyzing them off-line. This is less effective for multiprocessing systems for many reasons. Synchronization between threads can be sensitive to small changes in timing, so recording the traced data can change the flow of control of nondeterministic codes. The quantity of data in a multiprocessor is necessarily greater than a serial code and consumes more space, affecting locality. Finally, the multiplexing of threads to processors requires tools carefully integrated with the thread system to perform other than gross statistical analysis of behavior. For example, context switches must be recorded in order to determine which thread is responsible for particular cache misses.

Many microprocessors now have integrated performance counters that can be used to count events such as cache misses without substantially impacting code [53, 90]. Unfortunately, few operating systems make these available to user programs. User-level performance counters would make it much easier to characterize the impact of zones. More runtime information would allow new development tools. For example, individual zones could be evaluated for their effectiveness as part of profiling; it may also be possible to suggest potential zonings to the programmer based on heap analysis. If runtime environments could be made rich enough to determine the locality of objects and threads dynamically, the performance benefits of zones might even be possible without requiring any program annotations.

Libraries

Using Sather 1.1 language primitives, higher level libraries can be constructed to perform scheduling and mapping. Gomes [44] provides libraries that take as input a hierarchical dataflow description of neural nets. Through a combination of programmer estimation and trial runs, the libraries are able to produce a schedule and assignment of computation to nodes that respects communication requirements as well as load balancing.

The tree representation of zones is very general and convenient, but lacks expressiveness. For example, scheduling dependencies are often known in greater detail - for example, because the code is expressed as the evaluation of a data-flow graph, or through compile-time analysis of the synchronization constructs in a program [100]. It might be especially profitable to combine compilation techniques to avoid synchronization with scheduling and placement of locality trees. This also suggests that the implementation of zones be carefully integrated into thread and lock management.

Appendix: The Sather Language

This appendix presents Sather, an object-oriented language built on by the preceeding chapters. A brief language overview is given, with elaboration only of language features potentially unfamiliar because they have no analogue in C++. The threaded, synchronization and distributed extensions are fully specified; the distributed extension is also presented separately on page 47.

Coverage of Sather is useful to this work for several reasons. Sather is representative of a large class of imperative languages, but was designed specifically with performance and programming in the large. The threaded and synchronization extensions enable parallel processing and were carefully codesigned with the serial language, forming an appropriate foundation for exploratory design of the zone distributed extension.

SATHER

Sather is an object oriented language designed to be simple, efficient, safe, and non-proprietary. It aims to meet the needs of modern research groups and to foster the development of a large, freely available, high-quality library of efficient well-written classes for a wide variety of computational tasks. This section briefly describes the Sather language and gives an overview of the ICSI implementation, which includes serial, parallel, and distributed implementations. The following sections expand implementation details relevant to portable and modular performance.

Language overview

Sather has adopted ideas from a number of other languages. It was originally envisioned as a cleaned-up, efficient version of Eiffel, but now incorporates ideas and approaches from many languages. One way of placing it in the 'space of languages' is to say that it

attempts to be as efficient as C, C++, or Fortran, as elegant but safer than Eiffel or CLU, and to support higher-order functions as well as Common Lisp, Scheme, or Smalltalk. Cecil, ML, Modula-3, Oberon, School, and Self also influenced the language design. Although Sather predates Java, the languages have striking similarities, such as garbage collection, dynamic dispatch, assertions, and strong typing. Sather additionally offers multiple inheritance, parameterized classes and higher performance than Java, but does not attempt to address Java's design goals of security or dynamic loading of code.

There have been two generations of Sather. The original language was most indebted to Stephen Omohundro with contributions by Chu-Cheow Lim, Heinz Schmidt, Jerome Feldman and Franco Mazzanti. The first parallel Sather compiler was implemented by Chucheow Lim on the Sequent Symmetry, Sun workstations and the CM-5. A group at the University of Karlsruhe under the direction of Gerhard Goos also created a compiler for Sather 0.1. The language their compiler supported later diverged from the ICSI specification.

Sather 1.0 was a major language change, introducing bound routines, iterators, proper separation of typing and code implementation, contravariant typing, strongly typed parameterization, exceptions, stronger optional runtime checks and a new library design. The new language design effort involved many people; the compiler was a completely fresh effort by Stephen Omohundro and David Stoutamire. The parallel language design was largely due to Stephan Murer and David Stoutamire.

The language used throughout this document is Sather 1.1, an incremental improvement to 1.0. That compiler is the work of David Stoutamire, Michael Philippsen, Claudio Fleiner, and Boris Vaysman. Unlike previous specifications, in 1.1 the parallel and distributed extensions present a shared memory abstraction to the programmer while allowing explicit placement of data and threads.

A few of the language features are touched on here. Only concepts important to Sather that the reader that are not familiar to users of C++ are covered. The intention is to provide context in which to place the description of implementation specific to zones; other sources exist for the language specification [88], tutorial [74], design [89], and evangelization [87].

Garbage collection and checking

Like many object-oriented languages, Sather is *garbage collected*, so programmers never have to free memory explicitly. The runtime system does this automatically when it is safe to do so. Idiomatic Sather applications generate far less garbage than typical Smalltalk or Lisp programs, so the cost of collecting tends to be lower. Sather does allow the programmer to manually deallocate objects, letting the garbage collector handle the remainder. With checking compiled in, the system will catch dangling references from manual deallocation before any harm can be done.

More generally, when checking options have been turned on by compiler flags, the resulting program cannot crash disastrously or mysteriously. All sources of errors that cause crashes are either eliminated at compile-time or funneled into a few situations (such as accessing beyond array bounds) that are found at run-time precisely at the source of the error.

No implicit calls

Sather does as little as possible behind the user's back at runtime. There are no implicitly constructed temporary objects, and therefore no rules to learn or circumvent. This extends to class constructors: all calls that can construct an object are explicitly written by the programmer. In Sather, constructors are ordinary routines distinguished only by a convenient but optional calling syntax. With garbage collection there is no need for destructors; however, explicit finalization is available when desired.

Sather never converts types implicitly, such as from integer to character, integer to floating point, single to double precision, or subclass to superclass. With neither implicit construction nor conversion, Sather resolves routine overloading (choosing one of several similarly named operations based on argument types) much more clearly than C++. The programmer can easily deduce which routine will be called.

In Sather, the redefinition of operators is orthogonal to the rest of the language. There is ''syntactic sugar'' for standard infix mathematical symbols such as '+' and '^' as calls to otherwise ordinary routines with names 'plus' and 'pow'. 'a+b' is just another way of writing 'a.plus(b)'. Similarly, 'a[i]' translates to 'a.aget(i)' when used in an expression. An assignment 'a[i] := expr' translates into 'a.aset(i,expr)'.

Separation of subtyping and code inclusion

In many object-oriented languages, the term 'inheritance' is used to mean two things simultaneously. One is *subtyping*, which is the requirement that a class provide implementations for the abstract methods in a supertype. The other is code inheritance (called *code inclusion* in Sather parlance) which allows a class to reuse a portion of the implementation of another class. In many languages it is not possible to include code without subtyping or vice versa.

Sather provides separate mechanisms for these two concepts. *Abstract classes* represent interfaces: sets of signatures that subtypes of the abstract class must provide. Other kinds of classes provide implementation. Classes may include implementation from other classes using a special 'include' clause; this does not affect the subtyping relationship between classes. Separating these two concepts simplifies the language considerably and makes it easier to understand code. Because it is only possible to subtype from abstract classes, and abstract classes only specify an interface without code, sometimes in Sather one factors what would be a single class in C++ into two classes: an abstract class specifying the interface and a code class specifying code to be included. This often leads to cleaner designs.

Bounded parametric polymorphism

Parametric polymorphism is the static parameterization of code by a type. This is what C++ calls templates; there is no equivalent in Java. Because of Sather's carefully designed *contravariant* rules for subtyping, it is possible to statically check parameterized classes for type safety no matter what types are later plugged in. This is essential for modular software engineering, because the writer of a parameterized class will never be able to test the class with every conceivable type plugged in by a client in advance. The contravariant subtyping rules also eliminate the need for implicit type checks at runtime, required in both Eiffel and Java.

Iterators

Early versions of Sather used a conventional 'until...loop...end' statement much like other languages. This made Sather susceptible to bugs that afflict looping constructs. Code which controls loop iteration is known for tricky "fencepost errors" (incorrect initialization or termination). Traditional iteration constructs also require the internal implementation details of data structures to be exposed when iterating over their elements.

Simple looping constructs are more powerful when combined with heavy use of *cursor* objects (sometimes called 'iterators' in other languages, although Sather uses that term for something else entirely) to iterate through the contents of container objects. Cursor objects can be found in most C++ libraries, and they allow useful iteration abstraction. However, they have a number of problems. They must be explicitly initialized, incremented, and tested in the loop. Cursor objects require maintaining a parallel cursor object hierarchy alongside each container class hierarchy. Since creation is explicit, cursors aren't elegant for describing nested or recursive control structures. They can also prevent a number of important optimizations in inner loops.

Experience with cursor objects led to the addition to the language of *iterators*, methods that encapsulate user defined looping control structures just as routines do for algorithms [71]. Code using iterators is more concise, yet more readable than code using the cursor objects needed in C++. It is also safer, because the creation, increment, and termination check are bound together inviolably at one point. Each class may define many sorts of iterators, whereas a traditional approach requires a different yet intimately coupled class for each kind of iteration over the major class. Sather iterators are part of the class interface just like routines.

Iterators act as a lingua-franca for operating on collections of items. Matrices define iterators to yield rows and columns; tree classes have recursive iters to traverse the nodes in pre-order, in-order, and post-order; graph classes have iters to traverse vertices or edges breadth-first and depth-first. Other container classes such as hash tables, queues, etc. all provide iters to yield and sometimes to set elements. Arbitrary iterators may be used together in loops with other code.

Closures

Sather provides higher-order functions through *method closures*, which are similar to closures and function pointers in other languages. These allow binding some or all arguments to arbitrary routines and iterators but defer the remaining arguments and execution until a later time. They support writing code in an applicative style, although iterators eliminate much of the motivation for programming that way. They are also useful for building control structures at run-time, for example, registering call-backs with a windowing system. Like other Sather methods, method closures follow static typing and behave with contravariant conformance.

Immutable and reference objects

Sather distinguishes between reference objects and immutable objects. Immutable objects never change once they are created. When one wishes to modify an immutable object, one is compelled to create a whole new object that reflects the modification.

Experienced C programmers immediately understand the difference when told about the internal representation the ICSI compiler uses: immutable types are implemented with stack or register allocated C 'struct's while reference types are pointers to the heap. Because of that difference, reference objects can be referred to from more than one variable (*aliased*), but immutable objects never appear to be. Many of the built-in types (integers, characters, floating point) are immutable classes. There are a handful of other differences between reference and immutable types; for example, reference objects must be explicitly allocated, but immutable objects 'just are'.

Immutable types can have several performance advantages over reference types. Immutable types have no heap management overhead, they don't reserve space to store a type tag, and the absence of aliasing makes more compiler optimizations possible. For a small class like 'CPX' (complex number), all these factors combine to give a significant win over a reference class implementation. Balanced against these positive factors in using an immutable object is the overhead that some C compilers introduce in passing the object on the stack.

Immutable classes are never strictly necessary; reference classes with immutable semantics work too. For example, the reference class 'INTI' implements immutable infinite precision integers and can be used like the built-in immutable class 'INT'. The standard string class 'STR' is also a reference type but behaves with immutable semantics. Explicitly declaring immutable classes allows the compiler to enforce immutable semantics and provides a hint for efficient code generation.

pSather

Parallel Sather (pSather) is a collection of parallel extensions to the serial language. They extend serial Sather with threads, synchronization, and data placement. The Sather 1.1 distributed extension allows the placement of threads and objects on specific clusters of processors. The zone extensions described in the chapter Zones (page 50) are alternate ways of expressing placement.

pSather differs from concurrent object-oriented languages that try to unify the notions of objects and processes by following the *actors* model [64]. There can be a grave performance impact for the implicit synchronization this model imposes on threads even when they do not conflict. While allowing for actors, pSather treats object-orientation and parallelism as orthogonal concepts, explicitly exposing the synchronization with new language constructs.

pSather follows the Sather philosophy of shielding programmers from common sources of bugs. One of the great difficulties of parallel programming is avoiding bugs introduced by incorrect synchronization. Such bugs cause completely erroneous values to be silently propagated, threads to be starved out of computational time, or programs to deadlock. They can be especially troublesome because they may only manifest themselves under timing conditions that rarely occur (*race conditions*) and may be sensitive enough that they don't appear when a program is instrumented for debugging (*heisenbugs*).

pSather makes it easier to write deadlock and starvation free code by providing structured facilities for synchronization. A *lock statement* automatically performs unlocking when its body exits, even if this occurs under exceptional conditions. It automatically avoids deadlocks when multiple locks are used together. It also guarantees reasonable properties of fairness when several threads are contending for the same lock. These semantics are obtained with extra runtime mechanisms not presented here, but discussed in [37, 44, 77].

Implementation overview

The initial Sather system (“version 0”) was written over the summer of 1990 and soon released for public use. The 1.0 compiler design, begun in 1993, was heavily influenced by lessons learned from the earlier compiler. Two of these are remarkable because they contradict conventional academic compiler design principles. The lessons were that the utility of compiler construction tools is sometimes overstated, and heavy object-orientation may obscure an overall compiler design.

The Sather 0 compiler was a blend of lex, yacc, C and Sather. This required excessive communication between language domains which created configuration woes, and hindered debugging. In contrast, the 1.0 compiler is entirely in Sather, aside from a small runtime in C. The scanner and parser are both hand-written and only a very small part of the compiler. The resulting simplicity more than makes up for the use of an imperative style for those

pieces. Automated tools make sense for prototyping while a language definition is changing, but this is not as important for production compiler code.

Another important lesson later applied to the 1.0 compiler was that it can be poor software engineering to over-use polymorphism. The 0.x compiler was written with a class for each syntactic construct, with many dispatched methods corresponding to stages of compilation. As a result, the code executing for a given phase and its data dependencies were scattered over many files and difficult to find or change. In the 1.0 compiler, code was instead grouped by functional stage. This creates a stronger typing, by separating objects that would persist over the entire compilation with fields only meaningful in some stages into distinct object classes with only the fields relevant to each stage of compilation.

A major goal of the ICSI compilers was portability. This led to compilation through ANSI C code. The use of C as a portable assembly language influenced other design decisions; for example, C compilers can be expected to perform basic optimizations such as constant folding and register allocation, so the compiler design was not directly concerned with performing those optimizations. On the other hand, an understanding of the limitations of C compiler optimizers guided the selection of optimizations that the Sather compiler does perform.

The compiler uses two primary intermediate representations, the abstract syntax and abstract machine. In the abstract syntax tree, each node corresponds to a syntactic Sather entity. Sather source files are scanned and parsed by recursive descent. For parsimony in small programs, library source files are parsed only as the classes they contain are required. From the abstract syntax trees, all types, parameterizations, and interfaces of types are constructed and verified for type conformance. Once this has been done it is possible to build the implementations of each class; this is a flattening of parameterizations by parameter specialization. Specializing each instance of a parameterized class is similar to the way C++ compilers handle templates, and provides many opportunities for optimization.

Translation

Each reachable method is translated from the abstract syntax to abstract machine form. The nodes of AM trees correspond closely to C constructs and high level runtime events such as maintaining the exception stack, allocating an object or initializing an iterator. Translation occurs method by method, depth first, starting with the routine `main`. Unreachable code is not translated.

Inlining, optimizations, and type checking of method calls also occur at this time. Every method called is translated before translation of that method begins. After translation of a method, the suitability of methods for inlining and type-guided side-effect information for optimizations occurs, so this information is available to the calling method when it is translated. Sather optimizations include:

- Inlining is performed on small routines and iterators. Many common iterators, such as those to traverse arrays, also have special C translations that avoid loop overhead and use pointer arithmetic. The ability to do such inlining is not universal in C compilers and painful when it crosses source file boundaries. Similarly, iterators operate on state associated with a loop that persists across individual iterator calls; this is too much to expect a C compiler to handle well.
- Common subexpression elimination is performed, as well as hoisting of loop invariant expressions. Because full type-guided side-effect information is available and all source files are compiled together, the analysis of the Sather compiler can be better informed than what a C compiler can do by alias analysis.
- Special optimizations are performed for many Sather idioms, such as reassignment of immutable object fields and hoisting of iterator arguments with single-evaluation semantics. Again, these are exploitations of Sather semantics that are stronger than that reflected in the generated C code. Emperically, it was observed that many C compilers do a poor job with C structs, requiring assistance from the Sather compiler to get values off the stack and into registers. Similarly, C semantics guarantees that offset of struct fields must reflect the declaration order, while Sather semantics allows the compiler to reorder the fields to minimize fragmentation.
- The weak consistency model of Sather allows code motion of loads and stores. These optimizations are most important for distributed Sather, in which the overhead for a remote memory event may be hundreds of instructions. Additional mechanisms such as software caching and optimizations of common synchronization patterns are also available [37].

C generation

Once the abstract machine form has been generated, it is converted to C. The code generator was designed to meet many constraints.

- The generated code has to be portable; strictly ANSI-compliant C is emitted.
- It must be possible to have symbolic information emitted for debugging. The compiler emits readable C structs, and C debuggers may be used as they are, with "#line" directives inserted in the generated code. The generated code must be readable on demand. In addition to indented code, optional explanatory comments are inserted and the mangling of the Sather namespace to C must be reasonable.
- Giant C files tend to break compilers, time out, or thrash systems to death. For example, some C compilers and assemblers require time and space much greater than linear in the source file size. To keep C compilation possible, multiple C files are generated.
- C compilation is the bottleneck in compilation, so it should be kept fast. This means that the generated files must be appropriate for a parallel make utility. In addition, to amortize startup overhead files must not be too small; more than one class must go in a file. Only header information actually needed by a C file is be generated.
- When possible, C compilation should be incremental. Most changes to programs are very small, and should be reflected by smaller compile times. This means that global headers are a bad idea: if they change, all C files must be regenerated. Code is clustered in the C files to keep most changes local, and the namespace mapping attempts to not propagate changes between C files when it can be avoided. The name mangling must avoid symbols with alternative meaning, such as "printf".

Many of these goals interfere with one another. For example, the name mangling can't be deterministic because a generated name might collide with a reserved identifier, but if it isn't deterministic, then it is possible for "name-space pressure" to change the mapping used in other files, breaking incremental compilation.

A collection of heuristics was arrived at after exhaustive experimentation. A separate namespace is managed for each C entity, such as a struct. C names are constructed deterministically from the Sather namespace (for example, the routine FOO::bar(BAZ) is mapped to "FOO_bar_BAZ". When namespace collisions occur or the mapping would collide with a forbidden identifier (such as "printf") an alternative name is generated by deterministically appending an integer which will resolve the collision.

Routines are clustered by the class they are in. Decisions about what files to create and which classes to place in them is deferred until all C code is generated. Code for each of the classes is then merged, attempting to create C files of approximately the same length. Header information is generated for each of the resulting files separately, and must be sorted into a canonical order while respecting the struct's topological order in order to guarantee the same order of generation for each compile.

To enable incrementality, a hash value is generated for each file which is compared against the previous hash of that file; files are overwritten only when they change. A make utility

can then recompile only the files which have changed. A more aggressive compiler design would attempt to recompute all of its internal information incrementally, such that it could output the C which had changed without generating the C and doing the comparison.

Runtime

Generated code is customized for the target mostly by hooking into a runtime customized for that target, and sometimes additionally emitting different code sequences. Much of the runtime customization is handled by the C preprocessor, so there is no execution penalty, although programs may be slower to compile. For example, Sather allows many kinds of optional runtime checks to allow catching errors such as overflow and access to destroyed objects. These are implemented by macros, themselves controlled by a compiler command line switch. Other customizable runtime facilities include the maintenance of data structures associated with exceptions, threads, fair synchronization, remote memory access and memory management. Some of these are discussed in the chapters Zones (page 50) and Extending Zones (page 81).

Two portable interfaces are used: *Brahma* abstracts low-level thread, synchronization, and active messages, while *Siva* builds on these facilities to provide memory management and is the low-level interface for zones. This organization is illustrated by figure 22. Brahma provides functionality for dealing with individual clusters and threads; however, pSather provides language mechanisms at a higher level. Functionality that is implemented by the Sather compiler and runtime on top of Brahma include the management of locks, thread migration, and remote memory access.

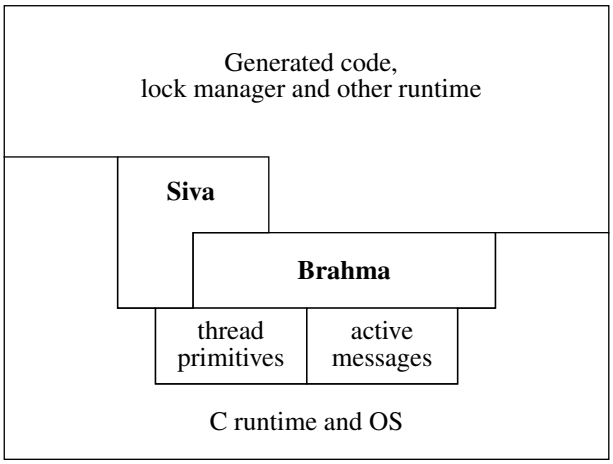


Figure 22: Dependencies between runtime components. Generated C code depends on Siva for memory management and Brahma for threads and messages, and is shielded from system-specific code by the Brahma and Siva interfaces.

Brahma

Sather provides locking constructs to atomically lock more than one lock, and permits sophisticated general locking conditions. Brahma synchronization only applies to individual, low-level locks. A distributed, high-level *lock manager* runtime is built on top of Brahma primitives. Sather lock statements are converted into manipulations of a runtime data structure that also manages exceptions; it is usually a stack, but can become a tree when a yield statement occurs inside of a protect statement. This data structure keeps all the history needed for resolution of multiple locks without deadlock or starvation. Sather lock classes are general and user-definable, allowing user extension of the lock manager.

There are three kind of synchronization provided by Brahma: there are mutual exclusions and semaphores, and a lightweight spin lock. The heavy versions are equivalent to the synchronization functionality in Solaris threads - they may block, and the critical regions they are used with may be of arbitrary length and be nested. The spinlock tries to use inline atomic instructions to build very fast spinlock-style mutual exclusion. This should only be used for very small, nonblocking, non-nested exclusion. The heavy versions can always safely replace a spinlock versions, with some performance penalty. Using locks and semaphores in request or reply handlers must be avoided to eliminate the possibility of deadlock.

Sather threads may be migrated between nodes with transparent context; Brahma threads are always local. Sather threads which move from node to node must be assembled from local threads. For example, when a remote call raises an exception, the exception must be caught remotely and re-raised on the initiating node. Similarly, the lock manager must be able to track the various Brahma thread IDs that a single Sather thread may assume in order to avoid deadlock. More details of the lock manager and inter-node thread management can be found in [37].

Brahma uses *active messages*, an asynchronous communication mechanism that is lightweight enough to expose the full hardware performance of modern interconnection networks [36]. Each message describes a user-level handler which is executed on message arrival with the message body as argument. The handler must execute quickly, and gets the message out of the network and into the computation as soon as possible. Active messages have constraints to avoid deadlock. Reply handlers must not block and it is only possible to reply to the same node that sends a request.

Tolerating latency requires overlapping communication and computation, which requires low-overhead asynchronous communication. The key optimization in active messages is the elimination of buffering. Eliminating buffering on the receiving end is possible because most messages hold a simple request to which the handler can immediately reply; for example, a handler servicing a request for a word of remote memory requires no storage. Buffering on the sending side is reduced because small messages can be released into the network quickly; the buffering in the network itself suffices.

As an optimization, active messages for setting up bulk memory transfers are provided. On some platforms, DMA and smart coprocessors can greatly reduce the overhead of this noncomputational operation. In Sather, these are invoked for large array copying between clusters.

Siva

Sather threads may transparently access memory on other nodes; Brahma provides a messaging facility, but no mechanisms for encoding pointers to a remote address space or resolving remote accesses. Similarly, garbage collection is inherently distributed, so Brahma can not address it. The Brahma interface views the world as a flat collection of address spaces; it is up to the client of the Brahma interface to add code to explicitly turn references from one address space to another into active message calls. The zone model, however, views the world as a hierarchy rather than a flat space. Siva makes use of Brahma to provide memory management customized to the memory hierarchy of the target.

Siva supplies a far pointer representation in which local pointers have unchanged representation while far pointers are encoded with unused bits. It is expected that the client code uses Brahma for communication and synchronization as well. Siva dictates the far pointer representation and must be used for memory management, but the Siva client (for Sather, the generated C code) is expected to translate pointers passed over messages explicitly. Brahma had to combine threads and active messages into a single package because they are not orthogonal. In the same way, Siva combines memory managment and the representation of far pointers into a single package because garbage collection needs to dictate the representation of pointers to be able to identify reachable objects.

On machines without 64 bit addresses, it is common to store far references by concatenating a node identifier and a pointer which is valid only within that node [79]. If normally unused high or low bits are used, this can be done using a single word. The pSather runtime uses a similar approach, with a special representation to make the near case fast. All references are represented relative to the node storing the pointer so that near pointers have the same representation as local pointers on that node. The bits of the address used to store the node identifier are all zero when the pointer is on that node. Code which composes messages translates from the representation of the pointer in the sending node to the representation appropriate in the receiving node; this requires two instructions with the Sparc ISA. In the common case of references to near variables, determining that the pointer is near only requires checking that the upper bits are zero, requiring two or three instructions.

Porting

Sather has been ported to a wide range of systems, enabled by compilation through ANSI C and the popular Boehm-Weiser conservative garbage collector [16]. Presently pSather runs under Solaris and Linux and has been ported to the Meiko MPP and a network of Myrinet connected four processor Sparcstation 10s. Table 15 shows the memory hierarchy

System	Level	Physical Connect	Repli-cation	Total Size	Block Size	Block count	Placement	Assoc-iativity
Intel PC: Pentium Pro	L1	chip	1	8K	32	128	H/W	2 way
	L2	module	1	256K	32	8K	H/W	4 way
	TLB	chip	1	256K	4K	64	H/W	4 way
	DRAM	board	1	64M	4K	16K	OS	Full
Ultrasparc I	L1	chip	1	16K	2x16	512	H/W	Direct
	L2	board	1	512K	64	16K	H/W	Direct
	TLB	chip	1	512K	8K	64	H/W	Full
	DRAM	board	1	128M	8K	16K	OS	Full
SS10: single Hypersparc	L2	module	1	256K	32	8K	H/W	Direct
	TLB	module	1	256K	4K	64	H/W	Full
	DRAM	board	1	128M	4K	32K	OS	Full
SS10 SMP: quad Hypersparc	L2	module	4	256K	32	8K	H/W	Direct
	TLB	module	4	256K	4K	64	H/W	Full
	Procs	board	1	1M	32	32K	H/W	4 way
	DRAM	board	1	128M	4K	32K	OS	Full
ICSI NOW: 3 nodes by Myrinet, quad Hypersparc	L2	module	12	256K	32	8K	H/W	Direct
	TLB	module	12	256K	4K	64	H/W	Full
	Procs	board	3	1M	32	32K	H/W	4 way
	DRAM	board	3	128M	4K	32K	OS	Full
	System	network	1	384M	128M	3	App	Full
Meiko MPP: 48 nodes by custom net, dual Hypersparc	L2	module	96	256K	32	8K	H/W	Direct
	TLB	module	96	256K	4K	64	H/W	Full
	Procs	board	48	512K	32	16K	H/W	2 way
	DRAM	board	48	128M	4K	8K	OS	Full
	System	network	1	6G	128M	48	App	Full

Table 15: Levels of the memory system of Sather platforms. All sizes in bytes; registers and disk storage are not shown.

for the systems to which pSather has been ported and used for performance results in this thesis. As always, quantitative analysis is recommended to help shed light on the subject [47].

THREADED EXTENSION

All Sather 1.1 implementations must support the language kernel, but the language extensions which may not be meaningful on every platform or which can be very difficult to implement. For example, the synchronization extension cannot be implemented without low-level thread support, and the distributed extension is not relevant on uniprocessors. Although these extensions are optional, they are part of the Sather specification. For example, Sather 1.1 implementations with thread support must provide the language extensions described here. Other language extensions not described here include standard interfaces to C and Fortran. The ICSI compiler supports all extensions described in the specification on one or more platforms.

In serial Sather there is only one thread of execution; in pSather there may be many. Multiple threads are similar to multiple serial Sather programs executing concurrently, but threads share variables of a single namespace. A new thread is created by executing a *fork*, which may be a par or fork statement (page 113), parloop statement (page 114), or an attach (page 118). The new thread is a *child* of the forking thread, which is the child's *parent*. pSather provides operations that can *block* a thread, making it unable to execute statements until some condition occurs. pSather threads that are not blocked will eventually run, but there is no other constraint on the order of execution of statements between threads that are not blocked. Threads no longer exist once they *terminate*. When a pSather program begins execution it has a single thread corresponding to the main routine.

Serial Sather defines a total order of execution of the program's statements; in contrast, pSather defines only a partial order. This partial order is defined by the union of the constraints implied by the consecutive execution order of statements within single threads and pSather synchronization operations between statements in different threads. As long as this partial order appears to be observed it is possible for a pSather implementation to overlap multiple operations in time, so a child thread may run concurrently with its parent and with other children. Using threads may render programs nondeterministic. Preconditions, postconditions, and class invariants may not work as intended when originally serial code is used with multiple threads.

The threaded extension may be implemented without the synchronization extension. This is only useful with data parallel code, in which it is not possible for threads to affect each other through side effects. Platforms may interpret such data parallelism in different ways, such as an opportunity for vectorization, or by executing only one 'thread' at a time.

par *and* fork *statements*

Example:

```
par
    fork ... end
end
```

Syntax:

fork_statement ⇒ fork *statement_list* end

par_statement ⇒ par *statement_list* end

Threads may be created with the *fork statement*, which must be syntactically enclosed in a *par statement*, which also implicitly creates a thread. When a fork statement is executed, it forks a *body thread* to execute the statements in its body. Local variables that are declared outside the body of the innermost enclosing par statement are shared among all threads in the par body. All threads created by a fork must complete before execution continues past the par. The rules for memory consistency apply to body threads, so they may not see a consistent picture of the shared variables unless they employ explicit synchronization (page 122).

Each body thread receives a unique copy of every local declared in the innermost enclosing par body. When body threads begin, these copies have the value that the locals did at the time the fork statement was executed. Changes to a thread's copy of these variables are never observed by other threads. Iterators may not occur in a fork or par statement unless they are within an enclosed loop. 'quit', 'yield', and 'return' are not permitted in a par or fork body.

As a generalization of serial Sather, it is a fatal error if an exception occurs in a thread which is not handled within that thread by some protect statement. Because par and fork bodies are executed as separate threads, an unhandled exception in their bodies is a fatal error.

Par and Fork Examples

In this code A and B can execute concurrently. After both A and B complete, C and D can execute concurrently. E must wait for A, B, C, and D to terminate before executing.

```
par
    par
        fork A end;
        B
    end;
    fork C end;
    D
end;
E
```

In this code, 'outer' is declared outside the par, so this variable is shared by the forked thread. However, because 'inner' is inside the par, the fork body receives its own local copy at the time of the fork.

```
outer:INT;
par
    inner:INT;
    fork
        -- fork body
    end
end
```

parloop *statement*

Example:

```
parloop c::=clusters! do ... end
```

Syntax:

parloop_statement ⇒ parloop *statement_list* do *statement_list* end

The *parloop statement* is syntactic sugar to make convenient a common parallel programming idiom.

parloop *S1* do *S2* end

is syntactic sugar for:

par loop *S1* fork *S2* end end end

Parloop example

This code applies 'frobnify' using a separate thread for each element of an array.

```
par
    loop e::= a.elt!;
        fork e.frobnify end
    end
end
```

Using the parloop shorthand, the same code could also be written:

```
parloop e: := a.elt! do
    e.frobnify
end
```

SYNCHRONIZATION EXTENSION

The synchronization extension allows threads to block; this requires threading facilities not available on every platform. Programmers should not assume that synchronization is less expensive than thread creation; creating threads as required may be more efficient than attempting to manage a pool of threads that wait for things to do. Generally, minimizing synchronization provides the greatest throughput.

lock *statement*

Examples:

```
lock
    when m then ...
    else ...
end;
lock
    guard d.size > 0 when m then ...
    when rw.writer then ...
end
```

Syntax:

lock_statement ⇒
    lock *expression* { , *expression* } then *statement_list* [else *statement_list*] end
    lock *lock_when* { *lock_when* } [else *statement_list*] end

lock_when ⇒
    [guard *expression*] when *expression* { , *expression* } then *statement_list*

Locks are special built-in synchronization objects that control the blocking and unblocking of threads. A thread *acquires* a lock, then *holds* the lock until it *releases* it. A single thread may acquire a lock multiple times recursively; it will be held until a corresponding number of

releases occur. Exclusive locks, such as 'MUTEX', may only be held by one thread at a time. In addition to these simple exclusive locks, it is possible to lock on other more complex synchronization types (on page 117).

Locks may be safely acquired with the *lock statement*. Expressions following a 'when' or 'lock' are called *locking conditions*, and must be subtypes of $LOCK (page 117). The statement list following the 'then' is called the *lock branch*. A lock statement guarantees that all listed locks are atomically acquired before a lock branch executes. Expressions following a 'guard' are called *guarding conditions*. The statements following the 'else' are called the *else branch*. The 'when' is dropped in the first form, convenient when there is only a single locking condition and no guard.

When a lock statement is entered the following occur in strict order:

1. Any guarding conditions are evaluated in textual order. If any evaluate to 'false', the corresponding when clause will not be considered further. when clauses without a guarding condition or for which the condition evaluates to 'true' are *accepted*.
2. If no when clauses are accepted, the else branch executes; it is a fatal error if there is no else clause in such a case.
3. For all accepted clauses, all locking conditions are evaluated, in textual order, left to right.
4. If the locking conditions of some when clause can be immediately satisfied, those locks are obtained, the corresponding lock branch executes, and execution concludes without considering other accepted when clauses.
5. If there is an 'else' clause and no when clauses have lock conditions that can immediately be satisfied, then the else branch executes. If there is no 'else' clause, the executing thread blocks until the locking conditions of some when clause can be satisfied. After the locking conditions are locked atomically, the corresponding lock branch executes.

Because all listed locks are acquired atomically, deadlock can never occur due to concurrent execution of two or more lock statements with multiple locks, although it is possible for deadlock to occur by dynamic nesting of lock statements or through other synchronization.

The implementation of lock statements also ensures that threads that can run will eventually do so; no thread will face starvation because of the operation of the locking and scheduling implementation. Similarly, no when clause will be repeatedly chosen over another such that a clause starves; for each when clause that can be acquired, there is a nonzero probability that it will be chosen. However, it is frequently good practice to have threads whose programmer supplied enabling conditions are never met in a given run (exceptional cases) or are not met after some time (alternative methods). One thread in an infinite loop can prevent other threads from executing for an arbitrary time, unless it calls SYS::defer (page 123).

All locks acquired by the lock statement are released when the lock or else branch stops executing; this may occur due to finishing the branch, termination of a loop by an iterator, a return, a quit, or an exception. yield may occur in a lock statement, but locks are not released until the iterator quits. Exceptions in a lock body will not be raised outside the body until all associated locks have been released.

unlock statement

Example:

```
unlock g
```

Syntax:

unlock_statement ⇒ unlock *expression*

Locks may also be unlocked before exiting the lock branch by an unlock statement. An unlock statement must be syntactically within a lock branch; in a par or fork statement an unlock must be inside an enclosed lock branch. It is a fatal error if the expression does not evaluate to a $LOCK object which is locked by the enclosing lock statement.

$LOCK classes

All synchronization objects subtype from $LOCK. In addition to primitive $LOCK classes, some synchronization classes return $LOCK objects to allow different kinds of locking. The concrete type of the returned object is dependent on the pSather implementation.

- MUTEX is a simple mutual exclusion lock. Two threads may not simultaneously lock a MUTEX.
- RW_LOCK is used to manage reader-writer synchronization, and defines two methods 'reader' and 'writer'. These return $LOCK objects. If 'rw' is an object of type RW_LOCK, then a lock on 'rw.reader' or 'rw.writer' blocks until no thread is locking on 'rw.writer', although multiple threads can simultaneously hold 'rw.reader'. Readers are granted priority over writers. Attempting to obtain a writer lock while holding the corresponding reader lock causes deadlock.
- WR_LOCK and FRW_LOCK also manage reader-writer synchronization. WR_LOCK gives writes priority over reads, while FRW_LOCK grants readers and writers equal priority.
- Classes under $ATTACH and $ATTACH{T} (page 119) also subtype from $LOCK.

Locking example

This code implements five dining philosophers. The philosophers are seated at a round table and forced to share a single chopstick with each neighbor. They alternate between eating and thinking, but eating requires both chopsticks.

```
chopsticks ::= #ARRAY{MUTEX}(5);
loop chopsticks.set!(#MUTEX) end;
parloop
    i::= 0.upto!(4);
do
    loop
        think;
        lock
        when chopsticks[i], chopsticks[(i+1).mod(5)]
        then eat
        end
    end
end
```

Attach statement

Example:

```
g :- forked_computation
```

Syntax:

attach ⇒ *expression* :- *expression*

Threads can be created by executing an *attached*. The left side must be of type '$ATTACH' or '$ATTACH{T}'. If the left side is of type '$ATTACH{T}', the return type of the right side must be a subtype of 'T'. If the left side is of type '$ATTACH', the right side must not return a value. There must be no iterators on the right side.

When an attach is executed, the following takes place in strict order:

1. The left side is evaluated.
2. $ATTACH and $ATTACH{T} both subtype from $LOCK. If the synchronization object of the left side is locked by another thread, the executing thread is suspended until it becomes unlocked.
3. Any local variables on the right side are evaluated.
4. A new thread is created to execute the right side. This new thread is *attached* to the synchronization object of the left side. The new thread receives a unique copy of every local variable; changes to these locals by the originating thread are not observed by the new thread. Similarly, if 'out' and 'inout' arguments occur on the right side, changes to local variables are not be observed by the originating thread. The rules for memory consistency (page 122) apply for other variables such as object attributes.
5. When execution of the right side completes, the new thread terminates, *detaches* itself, and enqueues the return value or increments the counter, if appropriate.

Attached threads may be thought of as producers that enqueue their return value (or increment a counter) when they terminate.

Every pSather thread is attached to exactly one $ATTACH object; even the main routine is attached to an unnamed object. The thread executing a par statement implicitly creates an $ATTACH object and forks a thread to execute the body. The newly created thread, as well as all threads created by fork statements syntactically in the par body, are attached to this same unnamed object. The thread executing a par statement blocks until there are no threads attached to the object. This ensures that all threads created by a fork have completed before execution continues past the par.

$ATTACH classes

There are four built-in $ATTACH classes; all subtype from $LOCK. These classes all have an implicit locked status (unlocked, or locked by a particular thread) and a set of attached threads.

- FUTURE{T} provides a handle to the result of a computation. It is an error to attach more than one thread to a future at a time.
- ATTACH allows multiple threads to be attached, but does not allow return values.
- *Gates* are powerful synchronization primitives which generalize fork/join, mailboxes, semaphores, and barrier synchronization. There is a typed GATE{T} that has a queue of values which must conform to 'T', and an unparameterized class GATE with only an integer counter.

In addition to thread attachment, these classes support the operations listed in the following tables 16, 17, 18, and 19. Some operations are *exclusive*: these lock the gate before proceeding and unlock it when the operation is complete. The exclusive operations also perform imports and exports significant to memory consistency (page 122).

Signature	Description	Exclusive?
create:SAME	Make a new unlocked synchronization object with an empty queue or zero counter and no attached threads.	N/A
has_thread:BOOL	Returns true if there is an attached thread.	No
threads:$LOCK	Returns a lock which blocks until lockable and there is some thread attached; then it is locked. Holding this lock does not prevent the completion of attached threads.	No
no_threads:$LOCK	Returns a lock which blocks until lockable and there are no threads attached; then it is locked. Holding this lock does not prevent the attachment of threads by the holder.	No

Table 16: Operations supported by ATTACH, FUTURE{T}, GATE, and GATE{T}

Signature	Description	Exclusive?
get:T	Return head of queue without removing. Blocks until queue is not empty.	Yes
empty:$LOCK	Returns a lock which blocks until lockable and the queue is empty; then it is locked. Holding this lock does not prevent the holder from making the queue become not empty.	No
not_empty:$LOCK	Returns a lock which blocks until the gate is lockable and the gate's queue is not empty; then the gate is locked. Holding this lock does not prevent the holder from making the queue become empty.	No

Table 17: Operations supported by FUTURE{T} and GATE{T}

Signature	Description	Exclusive?
size:INT	Returns number of elements in queue.	No
set(T)	Replace head of queue with argument, or insert into queue if empty.	Yes
enqueue(T)	Insert argument at tail of queue.	Yes
dequeue:T	Block until queue is not empty, then remove and return head of queue.	Yes

Table 18: Operations supported only by GATE{T}

Signature	Description	Exclusive?
size:INT	Returns counter.	No
get	Blocks until counter is nonzero.	Yes
set	If counter is zero, set to one.	Yes
enqueue	Increment counter.	Yes
dequeue	Block until counter nonzero, then decrement.	Yes
empty:$LOCK	Returns a lock which blocks until lockable and the counter is zero; then it is locked. Holding this lock does not prevent the holder from making the counter become nonzero.	No
not_empty:$LOCK	Returns a lock which blocks until the gate is lockable and the gate's counter is nonzero; then the gate is locked. Holding this lock does not prevent the holder from making the counter become zero.	No

Table 19: Operations supported only by GATE

Attach examples

Using a *future*. The statement 'f :- compute' creates a new thread to do some computation; the current thread continues to execute. It blocks at 'f.get' if the result is not yet available.

```
-- Create a future of FLT
f: := #FUTURE{FLT};
f :- compute;
...
result := f.get;
```

Obtaining the first result from several competing searches. Unlike a future, a gate may enqueue multiple values. When one of the threads succeeds, its result is enqueued in 'g'. If the results of the other two threads are not needed, additional code would be needed to prematurely halt the other threads.

```
g :- search(strategy1);
g :- search(strategy2);
g :- search(strategy3);
...
result := g.dequeue;
```

sync *statement*

Example:

```
sync
```

Syntax:

sync_statement ⇒ sync

The *sync statement* allows barrier synchronization between threads attached to the same synchronization object. A thread executing a sync blocks until all threads attached to the same object are also blocking on sync (or have terminated).

Sync example

This code applies 'phase1' and 'phase2' to each element of an array, waiting for all 'phase1' before beginning 'phase2':

```
parloop e::= a.elt! do e.phase1 end;
parloop e::= a.elt! do e.phase2 end
```

This code does the same thing without iterating over the elements for each phase. A single thread is forked for each element. Each thread executes 'phase1', the sync, and 'phase2'. The thread executing the par waits for all threads to terminate before proceeding.

```
parloop e::= a.elt! do
    e.phase1;
    sync;
    e.phase2
end
```

Because local variables declared in the parloop become unique to each thread, the explicit sync can be useful to allow convenient passing of state from one phase to another through the thread's local variables, instead of using an intermediate array with one element for each thread.

Memory consistency

Threads may communicate by writing and then reading variables or attributes of objects. All assignments are atomic (the result of a read is guaranteed to be the value of some previous write); assignments to variables of immutable type atomically modify all attributes. Writes are always observed by the thread itself. Writes are not guaranteed to be observed by other threads until an *export* is executed by the writer and a subsequent *import* is executed by the reader, even if the writes were previously observed by the reading thread. Exports and imports may be written explicitly (page 123) and are also implicitly associated with certain operations:

An import occurs:	An export occurs:
In a newly created thread	In parent thread when a child thread is forked
On exiting a par statement (children have terminated)	By a thread on termination
On entering one of the branches of a lock statement	On entering an unlock, or exiting a lock
On exiting exclusive operations (page 119)	On entering exclusive operations
On completion of a sync statement	On initiation of a sync statement

This model has the property that it guarantees sequential consistency to programs without data races.

Memory consistency examples

This incorrect code may loop forever waiting for flag, print 'i is 1', or print 'i is 0'. The code fails because it is trying to use flag to signal completion of 'i:=1', but there is no appropriate synchronization occurring between the forked thread and the thread executing the par body. Even though the forked thread terminates, the modification of 'flag' may not be observed because there is no import in the body thread. Even if the modification to flag is observed, there is no guarantee that a modification to 'i' will be observed before this, if at all.

```
-- These variables are shared
i:INT;
flag:BOOL;
par
    fork
        i := 1;
        flag := true;
    end;
    -- Attempt to loop until change
    -- in 'flag' is observed
    loop until!(flag) end
    #OUT + 'i is' + i + '\n'
end
```

This code will always print 'i is 1' because there is no race condition (unlike the previous example). An export occurs when the forked thread terminates, and an import occurs when par completes. Therefore the change to 'i' must be observed.

```
i:INT; -- This is a shared variable
par
    fork i:=1 end;
end
#OUT + 'i is' + i + '\n'
```

SYS *class*

pSather extends the SYS class with the following routines:

Routine	Description
defer	Inform scheduler that this is a good time to preempt this thread.
import	Execute an import operation (page 122).
export	Execute an export operation (page 122).

DISTRIBUTED EXTENSION

This section introduces distributed constructs that allow the programmer to extend pSather code with explicit placement information for efficiency on distributed pSather implementations. Explicitly placing objects and threads does not affect the semantics of the

original code, but it is also possible to deliberately change the original flow of control (ie. using with-near on page 125). The distributed extension presented here is replaced entirely by the zone extensions described in chapters Zones (page 50) and Extending Zones (page 81).

The memory performance model of pSather has two levels. The basic unit of location in pSather is the *cluster*. The programmer may assume that reading or writing memory on the same cluster is significantly faster than on a remote cluster. A cluster corresponds to an efficient group in the memory hierarchy, and may have more than one processor. For example, on a network of workstations a cluster would correspond to one workstation, although that workstation may have multiple processors sharing a common bus. This model is appropriate for any machine for which local cached access is significantly faster than general access.

At any time a thread has an associated *cluster id* (an INT), its *locus of control*. Until modified explicitly, the locus of thread remains the same throughout the thread's execution. When execution begins, the main routine is at cluster zero. The locus of control of a child thread is the same as the locus of its parent at the time of the fork.

The '@' operator

Example:

```
start_work @ least_loaded;
```

Syntax:

expression ⇒ *expression* @ *expression*

fork_statement ⇒ fork @ *expression* ; *statement_list* end

parloop_statement ⇒ parloop *statement_list* do @ *expression* ; *statement_list* end

The locus of a thread may be explicitly moved for the duration of the evaluation of a method call. An expression following the '@' must evaluate to an INT, which specifies the cluster id of the locus of control the thread will be at while it evaluates the preceding method. Subexpressions of the left side are evaluated at the current locus of execution and are not relocated. It is a fatal error for a cluster id to be less than zero or greater than or equal to clusters (see page 48). The '@' operator has lower precedence than any other operator. When iterator calls are on the left side, each iterator evaluation may be placed differently on successive iterations.

The '@' notation may also be used to explicitly place forked body threads of fork and parloop statements. Although for these constructs the location expression may appear to be within the body, the location expression is executed before threads are forked and is *not* part of the body.

Location expressions

All reference objects have a unique associated cluster id, the object's *location*. When a reference object is created by a thread, its location will be the same as the locus of control when the new expression was executed. A reference object is *near* to a thread if its current location is the same as the thread's locus of control, otherwise it is *far*.

There are several built-in expressions for location:

Expression	Type	Description
here	INT	The cluster id of the locus of control of the thread.
where(*expression*)	INT	The location of the argument. If the argument is void or an immutable type, it returns 'here'.
near(*expression*)	BOOL	true if the argument is on the same cluster as the executing thread. If the argument is void or an immutable type, it returns false.
far(*expression*)	BOOL	true if the argument is not on the same cluster as the executing thread. If the argument is void or an immutable type, it returns false.
clusters	INT	Number of clusters. Although a constant, may not be available at compile time.
clusters!	INT	Iterator which returns all cluster ids in order, 0 through clusters-1.

with-near *statement*

Example:

```
with able, baker near ... end
```

Syntax:

with_near_statement ⇒
    with *ident_or_self_list* near *statement_list* [else *statement_list*] end

ident_or_self_list ⇒ *identifier* | self { , *identifier* | self }

The *with-near statement* asserts that particular reference objects must remain near at runtime. The *ident_or_self_list* may contain local variables, arguments, and self; these are called *near variables*. When the with statement begins execution, the identifiers are checked to ensure that all of them hold either objects that are near or void. If this is true then the statements following near are executed, and it is a fatal error if the identifiers stop holding either near objects or void at any time. It is a fatal error if some identifiers hold neither near objects nor void and there is no else. Otherwise, the statements following the else are executed.

References

[1] Adler, Micah; Byers, John; and Karp, Richard. **Parallel Sorting With Limited Bandwidth.** *7th ACM Symposium on Parallel Algorithms and Architectures*, 1995.

[2] Aiken, Alexander; Fahndrich, Manuel; and Levien, Raph. **Better Static Memory Management: Improving Region-Based Analysis of Higher-Order Languages.** *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation,* p. 174-185. La Jolla, California, June 1995.

[3] Alpern, B., Carter, L., and Ferrante, J. **Modeling Parallel Computers as Memory Hierarchies.** Programming Models for Massively Parallel Computers, Giloi, W. K., S. Jahnichen, and B. D. Shriver ed., IEEE press, September 1993.

[4] Alpern, B., Carter, L., Feig, E., and Selker, T. **The Uniform Memory Hierarchy Model of Computation.** *Algorithmica*, 1994.

[5] Alpern, B. and Carter, L.. **Towards a Model for Portable Parallel Performance: Exposing the Memory Hierarchy.** Portability and Performance for Parallel Processing, Hey T. and J. Ferrante ed., John Wiley and Sons, 1994.

[6] Alpern, B., Carter, L., and Ferrante, J. **Space-Limited Procedures: A Methodology for Portable High-Performance.** International Working Conference on Massively Parallel Programming Models, 1995.

[7] Amza, Cristiana et al. **TreadMarks: Shared Memory Computing on Networks of Workstations.** *IEEE Computer*, Feb 1996, p. 18-28.

[8] Anderson, Ed et al. **LAPACK Users' Guide, Second Edition.** On-line at http://www.netlib.org/lapack, September 1994.

[9] Anderson, Eric. **Cache-Friendly Lists.** UCB CS264 class project, on line at http://HTTP.CS.Berkeley.EDU/~eanders/264/, Spring 1995.

[10] Austin, Todd. **Hardware and Software Mechanisms for Reducing Load Latency.** Ph.D. dissertation, University of Wisconsin - Madison, April, 1996.

[11] Bacon, David; Graham, Susan and Sharp, Oliver. **Compiler Transformations for High-Performance Computing.** Computer Science Division, University of California, Berkeley UCB/CSD-93-78, 1993.

[12] Bellosa, Frank and Steckermeier, Martin. **The Performance Implications of Locality Information Usage in Shared-Memory Multiprocessors.** *Journal of Parallel and Distributed Computing* 37, p. 113-121 1996.

[13] Bohr, Mark. **Interconnect scaling - the real limiter to high performance ULSI.** *Solid State Technology,*pp. 105-111, September 1996.

[14] Bershad, B., Lazowska, E. and Levy, H. **PRESTO: A System for Object Oriented Parallel Programming.** *Software: Practice and Experience*,18(8):713-732, August 1988.

[15] Bershad, Brian et al. **Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches.** *ASPLOS '94,* p. 158, 1994.

[16] Boehm, Hans. **Space Efficient Conservative Garbage Collection.** *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation,* p. 197-206. Albuquerque, New Mexico, June 1993.

[17] Bonwick, J. **The Slab Allocator: An Object-Caching Kernel Memory Allocator.** *Proceedings of the Summer 1994 USENIX Technical Conference*,p. 87-98, June 1994.

[18] Briggs, William. **A Multigrid Tutorial.** Lancaster Press, Lancaster, Pensylvania, 1994.

[19] Burger, Douglas; Hyder, Rahmat; Miller, Barton; and Wood, David. **Paging Tradeoffs in Distributed-Shared-Memory Multiprocessors.** *Supercomputing '94*, November 1994.

[20] Burger, Douglas; Goodman, James; Kagi, Alain. **The Declining Effectiveness of Dynamic Caching for General Purpose Microprocessors.** Univ. of Wisconsin-Madison CS TR 95-1261, 1995.

[21] Burger, Douglas; Goodman, James; Kagi, Alain. **Quantifying Memory Bandwidth Limitations of Current and Future Microprocessors.** *Proc. 23rd Intl. Symp. on Computer Architecture*, May, 1996.

[22] Cao, Pei; Felten, Edward; Karlin, Anna; and Li, Kai. **Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching, and Disk Scheduling.** *ACM Transactions on Computer Systems,*Vol. 14, No. 4, p. 311-343, November 1996.

[23] Carr, Steve et al. **Compiler Optimizations for Improving Data Locality.** *ASPLOS '94,* p. 252, 1994.

[24] Carter, Larry; Ferrante, Jeanne; Hummel, Susan; Alpern, Bowen; and Gatlin, Kang-Su. **Hierarchical Tiling: A Methodology for High Performance.** UCSD computer science department technical report CS96-508, November 1996.

[25] Carter, John; Bennett, John; and Zwaenepoel, Willy. **Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems.** Rice University Dept. Computer Science. Available at: ftp://cs.rice.edu/public/munin/tocs93.ps.Z, 1993.

[26] Chan, Tony. **Hierarchical Algorithms and Architectures for Parallel Scientific Computing.** *Proceedings of the ACM International Conference on Supercomputing*,p. 318-329, Amsterdam, The Netherlands, June 1990.

[27] Chandra, Rohit; Gupta, Anoop; Hennessy, John. **Data Locality and Load balancing in COOL.** *Fourth ACM Symp. on Prin. and Prac. of Parallel Prog.* p. 249-259, May 1993.

[28] Chandra, Rohit et al. **Scheduling and Page Migration for Multiprocessor Compute Servers.** *ASPLOS '94,* p. 12, 1994.

[29] Chandra, Rohit et al. **COOL: An Object-Based Language for Parallel Programming.** *IEEE Computer,* p. 13-26, August 1994.

[30] Coleman, Stephanie and McKinley, Kathryn. **Tile Size Selection Using Cache Organization and data Layout.** *Proceedings of the SIGPLAN '95 Conf. Programming Language Design and Implementation* pp. 279-290. La Jolla, California, June 1995.

[31] Cmelik, Robert. **Shade: A Fast Instruction-Set Simulator for Execution Profiling.** Univ. Washington tech report UWCSE 93-06-06, 1993.

[32] Culler, David et al. **Parallel Programming in Split-C.** Available at ftp://ftp.cs.berkeley.edu/ucb/CASTLE/Split-C/sc93.ps.Z, 1993.

[33] Culler, David et al. **LogP: Towards a Realistic Model of Parallel Computation.** *Proceeding of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[34] Dongarra, Jack et al. **MPI: A Message-Passing Interface Standard.** On-line at http://www.mcs.anl.gov/mpi/, June, 1995.

[35] Douglis, Fred. **The Compression Cache: Using On-line Compression to Extend Physical Memory.** *Proceedings of the Winter 1993 USENIX Conference*,p. 519-529, January 1993.

[36] von Eicken, Thorsten; Culler, David; Goldstein, Seth; Schauser, Klaus. **Active Messages: a Mechanism for Integrated Communication and Computation.** UCB Comp. Sci report UCB/CSD 92/675, March 1992.

[37] Fleiner, Claudio. **Parallel Optimizations, Advanced Constructs and Compiler Optimizations.** Ph.D. Thesis #1148, Institute of Informatics, University of Fribourg, Switzerland, 1997.

[38] Fowler, Robert and Kontothanassis, Leonidas. **Improving Processor and Cache Locality in Fine-grain Parallel Computations using Object-Affinity Scheduling and Contiuation Passing.** Univ. Rochester. Comp. Sci. Dept.Tech report 411, June 1992.

[39] Fraser, Christopher; Hanson, David. **A Code Generation Interface for ANSI C.** Available on-line, unknown date.

[40] Gamsa, Benjamin. **Region-Oriented Main Memory Management in Shared-Memory NUMA Multiprocessors.** Univ. Toronto Dept. Comp. Sci. TR-92-10-01, 1992.

[41] Geist, Al et al. **PVM 3 User's Guide and Reference Manual.** ORNL, May, 1993.

[42] Gharachorloo, Kourosh; Gupta, Anoop; and Hennessy, John. **Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors.** *ACM SIGPLAN Notices* 26(4):245-257, April 1991.

[43] Gifford, David et al. **Report on the FX-91 Programming Language.** MIT technical report MIT/LCS/TR-531, February 1992.

[44] Gomes, Benedict. **Mapping Connectionist Networks onto Parallel Machines: a Library Approach.** UC Berkeley PhD thesis, in progress.

[45] Grunwald, Dir; Zorn, Benjamin; and Henderson, Robert. **Improving the Cache Locality of Memory Allocation.** *SIGPLAN '93 Conf. on Programming Language Design and Implementation*, June 1993.

[46] Gwennap, Linley. **Digital 21264 Sets New Standard.** *Microprocessor Report,*Vol. 10 No. 14, pp. 11-16, October 28, 1996.

[47] Hamburgen, Bill et al. **Characterization of Organic Illumination Systems.** DEC Western Research Laboratory technical note TN-13, April, 1989.

[48] Hammond, K. et al. **Spiking Your Caches.** Glasgow FP tech report, 1994.

[49] Hayes, Barry. **Key Objects in Garbage Collection.** Ph.D. Thesis, Stanford University, March 1993.

[50] Hill, Mark and Larus, James. **Cache Considerations for Multiprocessor Programmers.** *Communications of the ACM,*Vol 33. No. 8, August 1990.

[51] Hill, Mark et al. **Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors.** *ACM Trans. Comp. Sys.*Vol. 11 No. 4 p. 300-318, November 1993.

[52] Hutchison et al. **The Emerald Programming Language.** Univ. Washington Dept. Comp. Sci. Tech. Report 87-10-07, October 1987.

[53] Intel. **Intel Architecture Software Developer's Manual.** On-line at http://www.intel.com, 1996.

[54] Jeremiassen, Tor and Eggers, Susan. **Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations.** *Proceedings of the 7th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, July 1995.

[55] Johnson, Kirk; Kaashoek, M. Frans; and Wallach, Deborah. **CRL: High-Performance All-Software Distributed Shared Memory.** *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, December 1995.

[56] Kane, Gerry and Heirich, Joe. **MIPS RISC Architecture.** Prentice Hall, 1992.

[57] Keleher, Pete; Dwarkadas, Sandhya; Cox, Alan; and Zwaenepoel, Willy. **TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems.** Rice University Dept. of Computer Science TR93-214. Also at: ftp://cs.rice.edu/public/munin/tr93-214.ps.Z, November 1993.

[58] Khalidi, Yousef; Bernabeu, Jose; Matena, Vlada; Shirrif, Ken; and Thadani, Moti. **Solaris MC: A Multi Computer OS.** *Proceedings of the 1996 USENIX Conference*, January 1996.

[59] Krishnamurthy, Arvind and Yelick, Kathy. **Optimizing Parallel Programs with Explicit Synchronization.** *Proceedings of the SIGPLAN '95 Conf. Programming Language Design and Implementation*pp. 196-204. La Jolla, California, June 1995.

[60] Larus, James. **Compiling for Shared-Memory and Message-Passing Computers.** *ACM Letters on Prog. Lang. and Systems,* Vol. 2 No. 1-4 p. 165-180, March-December 1993.

[61] Larus, James et al. **LCM: Memory System Support for Parallel Language Implementation.** *ASLPOS '94,* p. 208, 1994.

[62] Lebeck, Alvin R. and Wood, David A. **Cache Profiling and the SPEC Benchmarks: A Case Study.** *IEEE Computer*, October 1994.

[63] Lenoski, Daniel et al. **The Stanford DASH Multiprocessor.** *IEEE Computer,* p. 63-79, March 1992.

[64] Lim, Chu-Cheow. **A Parallel Object-Oriented System for Realizing Reusable and Efficient Data Abstractions.** International Computer ScienceInstitute technical report TR-93-063. Also available at http://http.icsi.berkeley.edu/Sather/tr-93-063.ps, 1993.

[65] Markatos, Evangelos and Chronaki, Catherine. **Trace-driven Simulation of Data Alignment and Other factors Affecting Update and Invalidate based Coherent Memory.** FORTH inst. comp. sci. TR/030, July 1993.

[66] Martin, Dan and Cartwright, Keith. **Solving Poisson's Equation using Adaptive Mesh Refinement.** See http://barkley.ME.Berkeley.EDU/~martin/AMRPoisson.html, 1996.

[67] McCalpin, John. **STREAM: Measuring Sustainable Memory Bandwidth in High Performance Computers.** See http://www.cs.virginia.edu/stream/, 1996.

[68] McKee, Sally. **Compiling for Efficient Memory Utilization.** Workshop on Interaction between Compilers and Computer Architectures, *Second IEEE Symposium on High Performance Computer Architecture (HPCA-2),* San Jose, CA, January 1996.

[69] McKenny, P.E. and Slingwine, J. **Efficient Kernel Memory Allocation on Shared Memory Multiprocessors.** *Proceedings of the Winter 1993 USENIX Technical Conference,*p. 295-305, January 1993.

[70] McKusick, M.K. and Karels, M.J. **Design of a General-Purpose Memory Allocator for the 4.3BSD UNIX Kernel.** *Proceedings of the Summer 1988 USENIX Technical Conference,*p. 295-303, June 1988.

[71] Murer, Stephan; Omohundro, Stephen; Stoutamire, David; and Szyperski, Clemens. **Iteration Abstraction in Sather.** *ACM Trans. on Prog. Lang. and Sys.*Vol. 18 No. 1 p. 1-15, January 1996.

[72] Murer, Stephan; Omohundro, Stephen; Stoutamire, David; and Szyperski, Clemens. **Iteration Abstraction in Sather.** *ACM Trans. on Prog. Lang. and Sys.*Vol. 18 No. 1 p. 1-15, January 1996.

[73] Neely, Michael. **An Analysis of the Effects of Memory Allocation Policy on Storage Fragmentation.** MA Thesis, University of Texas, May 1996.

[74] Omohundro, Stephen. **The Sather Language: Efficient, Interactive, Object-Oriented Programming.** *Dr. Dobbs*, October 1993.

[75] Philippsen, Michael and Mock, Markus. **Data and Process Alignment in Modula-2*.** *AP'93 Intl. Workshop on Automated Dist. Mem. Parallelization, Automatic Data Distribution and Automatic Par. Perf. Prediction,* Saarbrücken, Germany, p. 141-149, March 1993.

[76] Proebsting, Todd. **Code Generation Techniques.** PhD Thesis, Univ. Wisconsin, Madison, 1992.

[77] Quittek, J. and Weissman, B. **Efficient Extensible Synchronization in Sather.** International Scientific Computing in Object-Oriented Parallel Environments, first international conference (ISCOPE 97), December 1997.

[78] Rinard, Martin et al. **Jade: A High-Level, Machine-Independent Language for Parallel Programming.** *IEEE Computer*, June 1993.

[79] Rogers, Anne and Carlisle, Martin. **Supporting Dynamic Data Structures on Distributed Memory Machines.** *ACM Trans. on Prog. Lang. and Sys.* Vol. 17 No. 2 p. 233-263, March 1995.

[80] Saavedra, Rafael et al. **Characterizing the Performance Space of shared memory Computers Using Micro-Benchmarks.** Univ. of Southern California Dept. Comp. Sci.Tech report USC-CS-93-547, July 1993.

[81] Scales, Daniel et al. **Hierarchical Concurrency in Jade.** *Languages and Compilers for Parallel Computing*,Springer-Verlag LNCS 589, January 1992.

[82] Scales, Daniel and Lam, Monica. **An Efficient Shared Memory Layer for Distributed Memory Machines.** Stanford Tech report CSL-TR-94-627, 1994.

[83] Sciver, J.V. and Rashid, R.F. **Zone Garbage Collection.** *Proceedings of the USENIX Mach Workshop*,p. 1-15, October 1990.

[84] Shao, Zhong et al. **Unrolling Lists.** *Proc. 1994 ACM Conf. on Lisp and Functional Programming*, Orlando, FL, page 185-195, June 1994.

[85] Sites, Richard and Perl, Sharon. **PatchWrx - A Dynamic Execution Tracing Tool.** Submitted for publication, unknown, October 1995.

[86] SPARC International. **The SPARC V9 Memory Model.** In "The SPARC Architecture Manual V9", ed. Weaver and Germond, 1994.

[87] Stoutamire, David and Kennel, Matt. **Sather Revisited: A High-Performance Free Alternative to C++.** *Computers in Physics*,Vol. 9, No. 5, p. 519-524, Sept/Oct 1995.

[88] Stoutamire, David and Omohundro, Stephen. **The Sather 1.1 Specification.** International Computer Science Institute technical report TR 96-012, August 1996.

[89] Stoutamire, David; Zimmermann, Wolf; and Trapp, Martin. **An Analysis of the Divergence of Two Sather Dialects.** International Computer Science Institute technical report TR 96-037, August 1996.

[90] Sun Microsystems. **UltraSPARC's Advanced Memory Structured Minimizing Memory Overhead for System Design.** Sun WWW page, September 1994.

[91] Sussman, Alan et al. **PARTI Primitives for Unstructures and Block Structured Problems.** *Computing Systems in Engineering*, Vol. 3, No. 4, p. 73-86, 1992.

[92] Szyperski, Clemens; Omohundro, Stephan; Murer, Stephan. **Engineering a Programming Language: The Type and Class System of Sather.** *Programming*

Languages and System Architectures, Springer Verlag, Lecture Notes in Computer Science 782 ed. Jurg Gutknecht pp. 208-227, also ICSI TR-93-064, November 1993.

[93] Thekkath, Radhika and Eggers, Susan. **Impact of Thread Placement on Multithreaded Architectures.** University of Washington Dept. of Computer Science and Eng., FR-35, unknown date.

[94] Thekkath, Radhika and Eggers, Susan. **The Effectiveness of Multiple Hardware Contexts.** *ASPLOS* San Jose, CA, p. 328-337, Oct. 1994.

[95] Uhlig, Richard et al. **Design Tradeoffs for Software-Managed TLBs.** *ACM Trans. of Comp. Systems,* Vol 12, No. 3, pp. 175-205, August 1994.

[96] Valhalia, Uresh. **Kernel Memory Allocation.** Chapter in*"Unix Internals: the new frontier"*,Prentice-Hall, ISBN 0-13-101908-2, 1996.

[97] Valiant, L. **A Bridging Model for Parallel Computation.** *Communications of the ACM,*Vol. 33 No. 8, p. 103-111, 1990.

[98] Wawrzynek, John et al. **SPERT-II: A Vector Microprocessor System.** *IEEE Computer,* vol. 29 no. 3 p. 79-86, March, 1996.

[99] Weihl, William et al. **PRELUDE: A System for Portable Parallel Software.** MIT Technical Report MIT/LCS/TR-519, October 1991.

[100]Weissman, Boris. **Active Threads: an Extensible and Portable Lieght-Weight Thread System.** International Computer Science Institute technical report TR-97-036, 1997.

[101]Wen, Chih-Po. **Portable Library Support for Irregular Applications.** PhD Thesis, UC Berkeley, 1995.

[102]Wilson, Paul. **Uniprocessor Garbage Collection Techniques.** Submitted to *ACM Computing Surveys,* available on-line at ftp.utexas.edu, unknown date.

[103]Wulf, Wm. and McKee, Sally. **Hitting the Memory Wall: Implications of the Obvious.** *Computer Architecture News* 23(1):20-24, March 1995.

[104]Zhang, Zheng and Torrellas, Josep. **Speeding up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching.** *22nd Annual International Symposium on Computer Architecture,*p. 188-199, June 1995.

[105]Zorn, Benjamin. **The Effect of Garbage Collection on Cache Performance.** University of Colorado Dept. Computer Science CU-CS-528-91, May 1991.

[106]Zorn, Benjamin. **The Measured Cost of Conservative Garbage Collection.** Univ. Colorado, Boulder tech report CU-CS-573-92, April 1992.