The Programmer's Guide to SPERT

Krste Asanović
David Johnson

The University of California at Berkeley
Berkeley, California

The International Computer Science Institute
1947 Center Street, Suite 600
Berkeley, California 94704

Chapter 1

Introduction

This document is an introduction to writing programs for, and running programs on, the SPERT neural network and signal processor accelerator board. It is primarily a tutorial, although it will inevitably be used as a reference as some of the information it contains is not available elsewhere.

The aim of "The Programmer's Guide to SPERT" is to provide enough information for an experienced programmer to develop and run programmers on a SPERT board. It is assumed that the reader has a knowledge of the following:

- High level programming languages such as C or C++
- Modern microprocessor architectures
- The basics of computer arithmetic
- Assembler language programming

This document is available either in printed form, as a Postscript file or as a set of HTML pages for online reference.

Chapter 2

What is SPERT?

SPERT is a low cost neural network and signal processing accelerator board for workstations. The heart of the SPERT system is the T0 chip, a single-chip fixed-point vector microprocessor. One T0 has been combined with a high speed memory system and interface circuitry to form the SPERT board – a high performance computational subsystem that complements the facilities provided by modern workstations.

SPERT was developed jointly by the Computer Science Department of the University of California at Berkeley (UCB) and the International Computer Science Institute (ICSI).

Chapter 3

Compiling and Running SPERT Programs

3.1 Requirements

In order to develop and run SPERT programs, the SPERT development environment needs to be installed on your system. Contact ICSI for details on obtaining the software distribution, and follow the accompanying instructions for installation. The installation instructions are normally in a file called `INSTALL` in the base directory of the distribution. You will also need access to a SPERT board installed in a workstation. Although under some circumstances it is possible to develop and run SPERT programs using just the simulators, running non-trivial programs at reasonable speed usually requires real SPERT hardware.

3.2 The SPERT Development Environment

SPERT programs are compiled on the host sytems using a version of the GNU C/C++ compiler, `gcc`, that has been modified and configured specifically for SPERT. The development environment also includes a Torrent assembler, based on the GNU assembler `gas`, again with additions to support the Torrent instruction set. There are also various binary utilities including a disassembler and library archiver. All the programming utilities use the standard Unix names with the addition of the prefix "`spert0-`". Below is a summary of some of the executables available.

spert0-gcc C compiler (also compiles C++ programs)

spert0-c++ C++ compiler

spert0-as Assembler

spert0-ld Linker

spert0-ar Library archiver

spert0-ranlib Library ordering utility

spert0-objcopy Object file copying utility

spert0-objdump Object file dumping utility (includes disassembler)

spert0-nm Utility for displaying symbol names

spert0-strip Utility for stripping debugging information

spert0-size Utility for displaying the size of executables

spert0-strings Utility for extracting text strings from an executable

spert0-c++filt Demangler for C++ mangled symbol names

spertrun Run a SPERT program on a SPERTboard

spertrun-isa Run a SPERT program on the ISA simulator

spertrun-rtl Run a SPERT program on the RTL simulator

3.3 Compiling C and C++ programs for SPERT

The SPERT development environment includes compilers that target the scalar MIPS core that is part of T0. C and C++ programs are compiled and linked using the programs `spert0-gcc` and `spert0-c++`. These "compiler drivers" in turn invoke one or more programs to do the actual work of compilation. Possible sub-processes may include invocations of the C/C++ preprocessor, a language-specific compiler, the SPERT assembler and finally the SPERT linker.

Note that `spert0-gcc` can be used to compile both C and C++ programs, as can `spert0-c++`. The only difference between `spert0-gcc` and `spert0-c++` is the default value for some arguments, such as the assumed source language and which libraries to link with by default. In many circumstances, the two programs can be used interchangably, even when compiling C++ programs. Throughout this document, except where noted, all comments about `spert0-gcc` will also apply to `spert0-c++`.

The syntax of the `spert0-gcc` command line is similar to many Unix C compilers. Some useful flags that are:

`-g` Include debugging information in the object file

`-O2` Perform maximum optimization

Like many compilers, `spert0-gcc` and `spert0-c++` use filename suffices to decide on the appropriate handling for files given on their command line. Some standard suffices and their interpretation are:

file.**c** C source code which must be preprocessed.

file.**cc** C++ source code which must be preprocessed.

file.**cpp** C++ source code which must be preprocessed.

file.**cxx** C++ source code which must be preprocessed.

file.**C** C++ source code which must be preprocessed.

file.**S** Assembler code which must be preprocessed.

file.**o** Relocatable object file.

file.**a** Object file archive.

e.g. To compile the C program `hello.c` to produce an executable `hello` with debugging symbols:

```
spert0-gcc hello.c -g -o hello
```

e.g. To compile the C++ program `hello2.cc` to produce a relocatable object file `hello2.o` with maximum optimization and the NDEBUG symbol defined:

```
spert0-gcc -c -O2 -DNDEBUG hello2.cc
```

For more details of command line options and compiler features see [3].

3.4 Using the SPERT assembler

Although the SPERT assembler, `spert0-as`, is available as a stand-alone executable, it is normally invoked via `spert0-gcc`. Typically, code written for the Torrent architecture, (and the MIPS architecture, on which it is based) assumes that assembler source files will be passed through the C preprocessor before being assembled. Using `spert0-gcc` to assembler files ensures that this happens, and also allows a mixture of C and assembler files to be compiled and linked on the same command line. Assembler source files that need preprocessing should use the suffix `.S` in the file name in order to be handled correctly by `spert0-gcc`. A suffix of `.s` can be used if preprocessing is not required (although this is unlikely).

For example:

```
spert0-gcc main.c subs.S -o prog
```

`spert0-gcc` accepts several options that are passed on to the assembler, the majority of which are the same as for the MIPS version of `gcc`. Most of these are not fully supported in `spert0-gcc` and consequently should not be used. The exceptions are:

-mips1 Use only instructions for level 1 of the MIPS ISA - i.e. do not use branch likely instructions.

-mips2 Use instructions for level 2 of the MIPS ISA. This is the default.

In addition, `spert0-gcc` and `spert0-as` support the following options that are specific to the Torrent architecture.

-mschedule Enable instruction scheduling in the assembler – see chapter 9 for more details. Using this option will effectively prohibit symbolic debugging.

-mnoschedule Disable instruction scheduiling in the assembler. This is the default.

3.5 Examining SPERT Object Files

There are several situations where it is useful to examine the object files produced as the result of compiling, assembling or linking. One common need for object file viewing is when using the scheduling assembler – by disassembling the output of `spert0-as`, the effectiveness of the scheduling can be ascertained and the source modified to improve the scheduling possibilites available to the assembler.

The `spert0-objdump` program can be used to examine both relocatable object files, which typically have a `.o` prefix, as well as SPERT executables. `spert0-objdump` can disassemble executable code as well as display the contents of data sections and the values of symbols. In the case of C++ programs, it is often useful to pass the output of `spert0-objdump` through the name C++ demangler, `spert0-c++filt`.

E.g. to dissassemble an executable:

```
spert0-objdump -d mlp | spert0-c++filt
```

For more details see [5].

Chapter 4

Running SPERT Programs

There are two ways in which SPERT programs can be run – either on a physical SPERT board or on a simulator. For production program runs, having access to SPERT hardware is essential, although for debugging a simulator may be adequate. The simulators and the actual hardware have almost identical funtionality except for issues of timing. Specifically, the simulators include all the operating system functionality of the SPERT board and, time permitting, can run any SPERT program.

4.1 Using `spertrun`

Running SPERT programs on a SPERT board is almost as simple as running a program on a Unix workstation. The syntax is:

> `spertrun` [*spertrun-options*] *spertprog* [*spertprog-args*]

Any arguments passed as *spertprog-args* are available to the main routine of the SPERT program just as they would be if the same program had been run on the host.

e.g. To run SPERT program `mlp` with command line arguments `-i file`

```
spertrun ./mlp -i file
```

The above example explicitly specifies that the `mlp` program is in the current directory. If no path is specified as part of the SPERT executable name, the `spertrun` command uses the environment variable `SPERTPATH` to search for a suitable executable. `SPERTPATH` is formatted the same way as the Unix `PATH` environment variable, and contains a colon spearated list of directory names. Note that `spertrun` also loads a small operating system program, `kernel.spert`, as well as the user program. Unless the path to the kernel is specified explicitly using the `-kernel` option, `kernel.spert` must also be accessible using the `SPERTPATH` variable.

`spertrun` uses a temporary file to prevent multiple simultaneous accesses to the SPERT board. An error message of the form:

```
spertrun(T0_boardlock): ERROR - the SPERT board on  \
device '/dev/spertsbus0' is already in use
SPERT in use, user 'davidj', process 11082, started \
Thu Feb 15 16:58:15 1996
```

indicates that another spertrun process is active. If you want your new SPERT job to execute after the current running job has finished, the `-wait` option can be given to `spertrun`. This can be used to create multiple pending SPERT jobs, although the requests are not stored in a queue and waiting jobs will execute in a random order. If the active SPERT job is owned by you, you can force it to be terminated using the `-force` option of the new `spertrun` command. If you simply want to enquire as to whether another SPERT job is executing, run the following:

```
spertrun -status x
```

In some exceptional situations, it is possible for a spurious lock file to remain despite a SPERT board being inactive. In these situations, simply check the `/tmp/` directory and delete the file with a name of the form `SPERT_15298.lock`.

4.2 SPERT Simulators

There are two SPERT simulators available, with the difference between them being the approach each uses to simulate the T0 processor.

The ISA (Instruction Set Architecture) simulator models the execution of each T0 instruction – it will execute any SPERT program, but cannot be used to estimate program execution time or effects due to caches and interlocks. Typically, the ISA simulator runs SPERT programs 100-1000 times slower than a SPERT board, depending on the amount of vectorized code being executed and the speed of the workstation performing the simulation. The name of the ISA simulator program is `spertrun-isa`, and it takes many of the same arguments as the `spertrun` program.

The RTL (Register Transfer Level) simulator models the microarchitecture of the T0 processor down to the level of individual latches. For code without interrupts and I/O, execution time of a program on a real T0 can be accurately predicted from a run of the program on the RTL simulator. Typically, the RTL simulator runs SPERT programs 50,000 times slower than a SPERT board. Unlike the ISA simulator, the execution time of the RTL simulator is proportional to the real execution time regardless of instruction mix. An additional advantage is that a detailed trace of processor activity can be produced, including information interlocks and cache misses. This can be extremely useful when optimizing Torrent assembler code – see chapter 9 for more details. The name of the RTL simulator program is `spertrun-rtl`, and once again its use is very similar to `spertrun`.

Chapter 5

Programming in C and C++

5.1 Memory Use

Memory use is more of an issue with SPERT than with a typical interactive computer system. This is because SPERT boards typically have less memory than a workstation – a normal SPERT board comes with 8 MB, as opposed to the 64 MB or more often found in even a bottom end workstation. This difference is emphasized by two more features of SPERT. First, there is no virtual memory on SPERT – when all memory is used, your program will crash, not just slow down through paging. Secondly, there is no memory protection in the T0 processor, meaning that when memory is exhausted, the symptoms of the demise of your program may vary depending on exactly how the memory limit was reached. Below is more detailed description of the memory allocation features of SPERT.

5.1.1 SPERT User Process Memory Organization

The memory organization of a SPERT process is a degenerate version of a typical Unix process memory organization.

First, a SPERT processes static memory areas are allocated starting from

Section name	Contents
`.text`	Program code
`.rdata`	Read only data
`.data`	Initialized data
`.lit4`	Initialized data (floats)
`.lit8`	Initialized data (doubles)
`.sdata`	Initialized data (accessed via `$gp`)
`.sbss`	Uninitialized data (accessed via `$gp`)
`.bss`	Uninitialized data

Table 5.1: SPERT Static Memory Areas

the beginning of the user memory area (0x80010000). Each region takes up as much space as necessary, with the following region starting immediately after the previous one (with maybe a little empty space to allow alignment to appropriate boundaries). Table 5.1 indicates the order of static memory areas. The end of the static data area is marked by the absolute symbol `_end`.

After the static memory areas comes the heap. This is used for dynamically allocated memory – typically created using `malloc()` in C or `new` in C++. When the heap is too small, its size is increased by the memory allocation routnies using the `sbrk()` function call. When this happens, a check is made to ensure there is at least 16 KB of free space between the top of heap and the bottom of the stack, and if this is not the case the memory allocation request fails. The end of the heap is pointed to by a word stored at location `_heapend`. On SPERT, the size of the heap never decreases, although there amount of unallocated memory within the heap may increase.

The exact organization of the heap is controlled by the C library memory allocation routines, `malloc()`, `free()` etc. To maximize efficiency and minimize fragmentation problems, these routines maintained several pools of different power-of-two sized memory blocks. One side effect of this is that it is impossible to allocate one memory block that is >= 1/2 of the size of available memory. This is because four bytes of overhead is added to the amount of memory requested, then this figure rounded to the next highest power of two size. Asking for half of free memory results in an attempt to allocate a block that is as big as all of memory, and consequently the operation fails.

After the heap comes an area of unused memory, and follwing the unused memory is the stack. The lower bound of the stack is pointed to by CPU register `%sp`, with the higher bound being at the top of available user memory (0x80800000 on an 8 MB system). The lower bound of the stack moves to numerically lower addresses as the stack grows with each function call, and to higher addresses as the stack shrinks when functions return.

5.1.2 Memory Exhaustion

Memory can be exhausted in two ways: either there is not enough free space to expand the heap or there is not enough room for a new stack frame on the stack.

The more simple of the two situations is running out of heap space. When the heap is exhausted, the requesting `malloc()` or `new` operation will fail. In the case of `new`, the program will terminate with an appropriate message (assuming the new operator has not been redefined). In the case of `malloc()`, `realloc()` etc., the function will return a `NULL` value which must be checked for by the user program. Running a program that does not test the return values of `malloc()` on /SPERT/ is asking for trouble!

Stack exhaustion is a more dangerous situation. Because the stack may grow whenever a function is called, it is both costly and difficult to check the available free memory whenever the size of the stack increases. Consequently, running out of space for the stack does not cause an immediate error. Instead, the stack and the heap will overlap, resulting in undefined results (typically exhibiting symptoms such as segmentation faults, divide by zeros and infinite loops!).

As stack memory overflow is so unpleasant, it is better to try and avoid it! Some techniques to avert stack overflow include:

- Checking the amount of free memory – see below.
- Avoid storing large objects on the stack, and instead store them on the heap. The efficient heap allocation algorithms mitigate the costs

```
unsigned long
spert_memfree()
{
    extern unsigned long _heapend;
    unsigned long stack;

    asm("move %0, $sp" : "=r" (stack));
    return stack - _heapend;
}
```

Figure 5.1: Calculating the Size of Unallocated Memory

associated with doing this.

- Avoid using `alloca()` (if you do not know what this is, do not it up in case you are then tempted to use it!).
- Do not use recursion unless completely necessary. If you have to use it, be especially careful about termination conditions to avoid excessive levels of nesting.

Note that because the `sbrk()` checks for 16KB of free space before it increases the heap size, it is necessary for the stack to use at least this much space between enlargements of the heap for a fatal stack overflow to occur.

5.1.3 Calculating Free Memory

Finding out the total amount of free memory is non-trivial because there is both the unused memory between the stack and heap and the free memory blocks on the stack to take into account.

To find out the amount of free memory between the stack and the heap, the routine in figure 5.1 may be used.

To find out the details of free space within the heap, the `mstats()` routine can be used. This is called with one `char*` argument – a string that is printed

```
Memory allocation statistics test1
free:    0 0 60 31 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
used:    0 0 4 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        Total in use: 1, total free: 0
```

Figure 5.2: Example Output from `mstats()`

to indicate the specific invocation of `mstats`. The result is a summary of heap usage sent to standard out – see figure 5.2 for an example.

Each column of numbers in the output form `mstats` represents the number of blocks that are in use and free for a given block size. The column at the left represents blocks of size 2^3 (i.e. 8 bytes, including a four byte overhead), with each successive column being for a power of two larger block. The "Total in use" and "Total free" values should refer to the total number of bytes in use and free, but some versions of the library (including that in spertdev version v0_04) have a bug that corrupts these values.

Chapter 6

Using the Vector Libraries

6.1 Fixed Point Arithmetic

Typically, computer arithmetic is divided into two types – fixed point arithmetic and floating point arithmetic. SPERT does not support floating point arithmetic in hardware, although IEEE standard floating point [6] has been implemented in software. However, at this stage, we will limit our discussion to fixed point aritmetic.

By the far the most common case of fixed point arithmetic in modern computers is integer arithmetic with a binary twos-complement representation. In this scheme, the bits in a word represent successive digits in a binary number, with an implied binary point to the right of the least significant bit. The most significant bit of the word is used as a twos complement sign bit. I.e. the value of an n bit integer word, v_i, is given by:

$$v_i = b_{n-1} \times (-2)^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0$$

This gives a range of expressable values of -2^{n-1} to $2^{n-1} - 1$. However, in signal processing where real-world values are being represented, it is often useful to cover alternative ranges by assuming that the implied point is at some other postion in the word. If numbers with a variety of ranges are used, the different binary points are catered for by assuming each number has an associated implied exponent, e. Conventionally, an exponent of 0 is assumed when the value is in the interval $[-1, 1)$. I.e. the fixed point value, v_x, of an n bit integer word with an implied exponent of e is given by:

$$v_x = b_{n-1} \times (-2)^e + b_{n-2} \times 2^{e-1} + \cdots + b_0 \times 2^{e-n+1}$$

To add two fixed point numbers with different exponents in an integer adder, it is necessary to shift one or other of the words to align their implied binary points. Multiplication may also require shifting the intermediate result value so that it has the binary point in the same position as the

6.2 Rounding and Saturation

6.3 Fixed Point Arithmetic in the Vector Libraries

Chapter 7

Programming in Assembler

Chapter 8

Debugging on SPERT

8.1 Crashes on SPERT

The SPERT runtime environment performs some rudimentary checks to ensure a SPERT program is executing correctly. Some errors will be spotted when they occur, causing the SPERT program to terminate immediately. Other forms of error will not halt the program, but will be reported after the program has terminated. In both cases, a copy of the SPERT memory image will be produced in the file `spert.core`.

8.2 Approaches to Debugging on SPERT

There are several approaches that can be used to debug SPERT programs.

- Using `printf` statements
- Monitoring system call activity
- Debugging using the `spert0-gdb` debugger
- Creating traces using the ISA simulator

These techniques vary in their applicability – different situations may require one or more approaches.

The "printf" style of debugging, although primitive, is often useful. The use of vectors and the presence of fixed point types combine to make many standard debugging approaches clumsy, especially when searching for bugs that reveal themselves in the form of arithmetic errors. In these cases, conversions of fixed point matrices to floating point or integer representation and then storage of all pertinent data in a file is often the best way to proceed.

8.3 Debugging Using Sytem Call Traces

When running SPERT programs using `spertrun` or an equivalent simulator program, the `-strace` command line option can be used to produce a trace of system call activity. The output is similar to that produced by the Unix `strace` command. This can be used to monitor program control flow at a gross level and to spot many simple programming errors. Typically system call traces are redirected to a file using the `-strace-file` option to `spertrun`.

An example of a system call trace is included below.

```
syscall: fstat(1, 807ffdf8) = 0
syscall: ioctl(1, TCGETA, 0x00000000) = 0
syscall: write(1, 0x80017c10, 12) = 12
```

All lines in a system call trace begin with the string "`syscall:`". This is followed by the name of the system call, its arguments and the resulting return value. Note that memory allocation system calls are normally not included in the trace.

8.4 Debugging Using `spert0-gdb`

`spert0-gdb` is a slightly modified version of the GNU debugger, `gdb`. The following changes from the "iris4" targetted version of gdb have been made:

- The default startup file is called `.spgdbinit`, as opposed to `.gdbinit`.
- The default prompt is `(spert-gdb)`.
- The value of the gdb *spertrun* variable is used to decide which server program to use when running the SPERT program. Use the gdb command `show spertrun` to view the value and `set spertrun` to set it. Typically, `.spgdbinit` will contain a line of the form:

  set spetrun spertrun

- The value of the gdb `spertrunflags` variable can be used to pass flags to the sever program. This can be used e.g. to use a different kernel. Note that arguments to the SPERT program are passed using the `gdb` *run* command.
- The addition scalar register variables `$vlr`, `$vcond`, `$vovf`, `$vsat` and `$vrev` have been added.
- The new command `info vregisters` (abbreviation `i vr`) can be used to view, but not modify, the T0 vector registers.

See [4] for more details on using `gdb`.

8.5 Debugging Using the ISA Simulator

The ISA simulator can produce traces of SPERT program execution in various formats. The `-t` option to `spertrun-isa` is used to turn on tracing, with the `-t-format` option being used to control exactly what trace information is emitted. The letters given in the `-t-format` argument form an ordered list of output specifiers, indicating what information should be printed each cycle and the order in which to display it. The following specifiers are useful:

c Cycle information - cycle number, PC and disassembled instruction of next instruction to execute.

r User scalar registers

v Vector registers

s System registers – useful for debugging the kernel

m Memory – use the `-t-mem-start` and `-t-mem-bytes` options to specify which area of memory

A typical format string would be "`rc`" – i.e. print the scalar registers followed by the disassembled instrucion. As the "`c`" specifier outputs the *next* instruction to execute, it is normally the last in the list.

Because the trace is typically extremely long, selecting a subset of all executed cycles is often appropriate. This can be done using `-t-start` and `-t-end` options to specifiy a range of cycle numbers to trace, with trial and error being used to select appropriate cycles. Alternatively, `-t-pc-low` and `-t-pc-high` can be used to select a range of program text to trace. The `-t-func` option works like `-t-pc-low` and `-t-pc-high`, but instead takes the name of a function, with the trace limits being set to the start and end of that function. This simply saves having to disassemble the executable to find the range of addresses to trace.

Note that producing traces has causes the ISA simulator to run at a fraction of its normal speed.

Chapter 9

Writing Efficient Torrent Assembler Routines

Bibliography

[1] Asanović, K., Johnson, D., “The Torrent Architecture Manual,” ICSI Technical Report, in Print.

[2] Kane, G., Heinrich, J., “MIPS RISC Architecture”, Prentice Hall, Englewood, Cliffs, NJ, 1992.

[3] Stallman, R., “Using and Porting GNU CC”, The Free Software Foundation, Boston, MA, 1995.

[4] Stallman, R., Pesch, R., “Debugging with gdb”, The Free Software Foundation, Boston, MA, 1995.

[5] Pesch, R., Osier, J., Cygnus Support, “The GNU Binary Utilities”, The Free Software Foundation, Boston, MA, 1995.

[6] “IEEE Standard for Binary Floating-Point Arithmetic”, IEEE, New York, 1985.