Dynamic Task Migration from SIMD to SPMD Virtual Machines

James B. Armstrong
Sarnoff Real Time Corporation
201 Washington Road
Princeton, NJ 08543-5300 USA

Howard Jay Siegel
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907-1285 USA

Abstract -- *A method to migrate a task dynamically from a virtual SIMD machine to a virtual SPMD machine is proposed. It is assumed that the SIMD and SPMD virtual machine models only differ to support the different modes of parallelism, and that the program was coded in a mode-independent programming language. The difficulty of performing an SIMD to SPMD migration arises from the fact that some SIMD PEs may be disabled at the point in the program at which the migration occurs. Furthermore, each of the disabled PEs may need to be re-enabled at different points in the SPMD program. The compiler information that should be furnished to the task migration procedure is given. The time and memory space complexities of the task migration procedure are also presented. This work solves part of the general problem of task migration in SIMD/SPMD mixed-machine heterogeneous systems.*

1. Introduction

In a heterogeneous system [9], different types of parallel machines are interconnected by high-speed links. Task migration in a such an environment may be necessary for fault-tolerance, load balancing, administrative reasons, or improving execution time of a single task. The migration procedure "captures" a program's execution state on one type of machine and then maps it to a viable state on a different type of machine. When task migration is performed in the context of fault-tolerance, the "capturing" of the state is done periodically at checkpoints, and the mapping is done at the time of the fault. The work described here pertains to task migration among parallel machines in all these contexts.

One possible approach to migrating dynamically a task between a synchronous single instruction stream – multiple data stream (SIMD) machine and an asynchronous single program – multiple data stream (SPMD) machine is illustrated in Fig. 1. In general, SPMD mode is the use of a multiple instruction stream – multiple data stream (MIMD) machine when all PEs execute the same program, but asynchronously with respect to one another. A task is assumed to be coded in some hypothetical mode-independent programming language [14], referred to here as the virtual programming language (VPL) (e.g., ELP [6]). The VPL compiler generates object code for each machine or a subset of machines on a network, as well as produces information necessary for the task migration procedure. The dotted arrows in Fig. 1 indicate that from a single VPL source program each machine's executable program is generated.

To move a task from some physical machine executing in one mode of parallelism to another physical machine executing in another mode of parallelism, two types of transformations are performed that rely on information generated by the VPL compiler. The first type of transformation (dashed arrows in Fig. 1) maps the execution state of the task on a particular machine to/from an execution state on a virtual machine model, which represents all machines with the same mode of parallelism. The first type of transformation would map a state on SPMD machines D, E, and F to/from a state on the virtual SPMD machine. Likewise, a state on SIMD machines A, B, and C would be mapped to/from a state on the virtual SIMD machine. The different shapes of the machines depict their differing physical architectures.

The second type of transformation (solid arrow in Fig. 1) migrates conceptually the task between a state on the virtual SIMD machine and a state on a virtual SPMD machine. The similarity of shapes between the virtual SIMD machine and the virtual SPMD machine represents that the conceptual architectures only differ to support the different modes of parallelism. This implies that, among other similarities, the number of processing elements in the virtual SIMD machine and that in the virtual SPMD machine are the same.

The delineation between the two types of transformations divides the task migration problem into manageable parts. It is possible that such a clear division of transformations will not exist when a general task migration procedure is actually implemented. The work described here addresses the SIMD to SPMD portion of the second type of transformation. It proposes a method by which a single instruction stream SIMD program can be mapped to a multiple instruction stream SPMD program. The approach taken is to characterize a single instruction stream program and a multiple instruction stream program as programs that execute on a virtual SIMD machine and virtual SPMD machine, respec-

James B. Armstrong was funded by a NASA Graduate Researchers Fellowship under grant number NGT-50961. This research was also supported by Rome Laboratory under contract number F30602-92-C-0150 and by NRaD under subcontract number 20-950001-70. Equipment supplied by the National Science Foundation under grant number CDA-9015696 was used.

0-8186-7123-8/95 $4.00 © 1995 IEEE

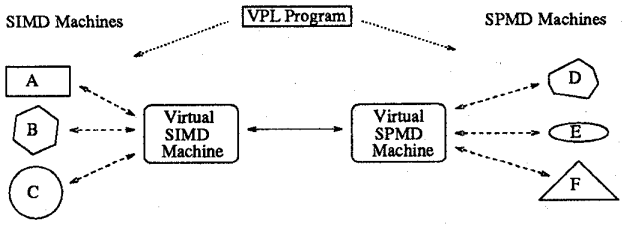

Fig. 1: Graphical depiction of task migration between SIMD and SPMD machines.

tively.

The point at which the SIMD program is migrated to an SPMD program on an SPMD machine may be obtained from a checkpointed state (i.e., if fault-tolerance is the context) or it may be an arbitrary point (i.e., if load balancing is the context). The difficulty of performing an SIMD to SPMD migration arises from the fact that some SIMD PEs may be disabled at the point in the program at which the migration occurs. Furthermore, each of the disabled PEs may need to be re-enabled at different points in the SPMD program.

There has been research done in migrating a task in the other direction, i.e., from an SPMD machine to an SIMD machine. The method described in [1] requires that a task be coded in an SIMD/SPMD mode-independent programming language, as does the work described here. A way to transform any MIMD program into pure SIMD code is presented in [2]. It does this by having the SIMD code (running on an SIMD machine) emulate the MIMD program. One of the goals of using a mode-independent language is to have the VPL compiler generate efficient code for each of the source and destination machines (i.e., the generated code is specifically targeted for each machine [14]). The task migration procedure maps a point in the efficient code for one machine to a point in the efficient code for another machine. Although the method proposed in [2] is a general method, the code executed on the destination machine may not be as efficient because the MIMD code is executed on an SIMD machine emulating MIMD mode.

Mechanisms for migrating tasks between different single-processor computers, which can be applied to specifying the first type transformation, have been proposed (e.g., [3], [4], [7], [11], [12], [13]). These papers were also useful for this study of the second type of transformation. Specifically, portions of that earlier work were helpful in defining the state of an executing program (used in Section 3). Furthermore, [13] defines the equivalence of points in two programs that are executing in a heterogeneous system, where each machine consists of a single CPU. Some aspects of that research, as it pertains to the work described here, can be simplified because of the system assumptions made here. However, other aspects are extended to account for parallel execution.

There is overhead associated with migrating a task from a SIMD to a SPMD machine that cannot be avoided. The compiler must perform extra computation to furnish the migration procedure with information necessary to migrate the task. In addition, this information requires extra memory storage. For many applications, the added burden of compiling a progam to allow dynamic task migration may not be necessary and should not be done. Instead, the program should just be restarted from the beginning on the new machine. However, it may be better to migrate the application in other cases, such as jobs of long duration. This paper describes a migration procedure that makes apparent the costs that must be present for a task to be migrated dynamically from a SIMD to a SPMD machine.

Section 2 states more specifically the assumptions about the programming language, operating system, and machine models as well as states formally (i.e., mathematically) the task migration problem. The procedure to migrate a task between an SIMD and an SPMD virtual machine is presented in Section 3.

2. Background and model

2.1. Overview

This section presents the underlying model and mathematical formulation that will be used throughout the paper. The virtual machine models (Subsection 2.2) and mathematical model (Subsection 2.3) were also used in [1], where the different problem of task migration from an SPMD machine to an SIMD machine was considered.

2.2. Virtual machine models

It is assumed that the virtual SIMD and SPMD machines differ only in the mechanism needed to support the different modes of parallelism. In these machines, each processor is paired with a memory module to form a PE (processing element), as most current commercial parallel systems are physically implemented (e.g., CM-2, CM-5, KSR1, MasPar MP-1, and nCUBE 2). The virtual SIMD machine is composed of a CU (control unit), P PEs, and an interconnection network. The enabled PEs receive and synchronously execute common instructions that are broadcast from the CU. Furthermore, the CU processor is assumed to have *Creg* general purpose registers available for storing data and each PE's processor is assumed to have *Sreg* general purpose registers. The virtual SPMD machine consists of P PEs and an interconnection network. Each PE's instructions and data are stored in its memory module. Because there are multiple threads of control, the PEs execute asynchronously with respect to one another. Also, each PE's processor is assumed to have $Creg + Sreg$ general purpose registers available for use.

The hypothetical VPL assumed here is a C syntax with extensions for parallelism, but, to simplify the migration procedure and without loss of program functionality, pointers to local variables and goto instructions are not allowed. VPL variables are either mono or poly [6]. A poly attribute specifies that a local copy of the variable resides in each PE in the machine

and each copy can have a different value among PEs anywhere in the program. A variable declared as mono has the same value across PEs at the same point in a program. In a SIMD machine, there is a single copy of the variable and is stored on the CU. In an SPMD machine, each PE has its own copy. Although, as an SPMD program executes, a mono variable may have different values across PEs at any point in time, it will always have the same value across PEs at the same program location (enforced by the compiler).

If a conditional statement consists of at least one poly variable, the conditional is considered to be a poly conditional statement, otherwise it is a mono conditional statement. In a SIMD machine, mono conditionals determine program control flow and poly conditionals enable/disable PEs. (It is assumed that each PE in an SIMD machine has an enable stack that stores the PE's enable status for various depth nestings of poly conditional expressions, e.g., MasPar MP-1, CM-2.) Each PE on an SPMD machine is able to test independently local mono or poly values and branch around code that should not be executed.

It is assumed that mono and poly variables share the same virtual address space in each machine; even though in SIMD mode the mono variables are physically stored in the CU and the poly variables are physically stored in the PEs. The mono and poly variables are stored in separate virtual memory segments to simplify the modification of virtual address tables (for both global static and global dynamically allocated variables). The mono and poly stack variables are assumed to occupy the same virtual memory segments, because, in general, SPMD machines would only have one stack pointer register in each PE. If pointers to mono and poly global variables are to be effectively handled, it is assume that each such variable has the same virtual (but not physical) address in each PE (or CU) in both virtual machines.

In SIMD mode, upon the call of a subroutine, stack space for mono parameters, mono local variables, and subroutines' return addresses is allocated on the CU. The memory space for poly parameters and poly local variables is allocated on the PE stack (e.g., MasPar MPL programming language [5]). In SIMD mode, it is assumed the CU and each PE has a frame pointer that points to the locally stored stack (e.g., PASM prototype). In SPMD mode, mono and poly parameters and local variables are pushed onto the PE stack. Stack and frame pointers are kept in each PE.

2.3. Mathematical model

To model the migration procedure mathematically, let a VPL program, F, be compiled to produce an object code program, S, for a virtual SIMD machine and an object code program, M, for a virtual SPMD machine. Either in SIMD mode (due to enabling/disabling) or SPMD mode (due to branching), not all PEs will necessarily execute the same sequence of instructions. $\hat{S}_{1:\sigma}(\mathbf{x})$ denotes the sequence of states representing the collective actions of all PEs that occur during the execution of the entire SIMD program with input $\mathbf{x}$, starting with state 1 and ending with state σ. $\hat{M}_{1:\mu}(\mathbf{x})$ is defined similarly for the SPMD program, when $\hat{S}_{1:\sigma} \equiv \hat{M}_{1:\mu}(\mathbf{x})$ for all $\mathbf{x}$ (i.e., the SIMD and SPMD programs produce the same result). This paper describes a transformation H_S, such that $\hat{M}_{j:\mu}(H_S(\hat{S}_{1:i}(\mathbf{x}))) \equiv \hat{M}_{1:\mu}(\mathbf{x})$, for $1 \le i \le \sigma$ and $1 \le j \le \mu$, and for all $\mathbf{x}$. The equivalence statement means that if $\hat{S}_{1:\sigma}$ is interrupted at some point having executed the sequence of states $\hat{S}_{1:i}$, then H_S can transform the results computed by $\hat{S}_{1:i}(\mathbf{x})$ to a form that can be processed by $\hat{M}_{j:\mu}$ (executed in SPMD mode), so that the result is the same as that found by $\hat{M}_{1:\mu}$.

To find interruptible points in both programs that are "equivalent," the VPL compiler divides the object code programs S and M into uninterruptible blocks of instructions with the following properties: (1) there are an equal number of blocks, B, generated from S and M, (2) the function implemented by block j from S and block j from M are equivalent for $1 \le j \le B$, and (3) no block in both S and M can be further divided into blocks such that properties (1) and (2) are true for the resulting blocks. By constructing programs S and M with uninterruptible blocks of instructions in this way, for any interruptible point in M there is an equivalent point in S.

2.4. Mapping information

There are several data structures that are assumed to be produced by the VPL compiler for each program that provide mapping information between S and M. The mappings between the memory addresses of subroutine call instructions and between interruptible points in the SIMD and SPMD programs are kept. This information is used by H_S to map equivalent points and return addresses in S to those in M, and is stored in a table called the ART (address resolution table). This table, therefore, provides a mapping from a block in one program to the equivalent block in another.

For nested poly conditional expressions in SIMD mode, two PEs may be in the disabled state at the same point in a program but will be enabled again at two different locations (because they were disabled at different points). To record these possible locations, the VPL compiler generates a static alternate block list (SABL) for each uninterruptible block inside a poly conditional statement. The elements of the SABL are defined based on the static nesting of conditionals within a given subroutine. The SABL contains all possible blocks in M to which a disabled PE may be mapped from the block in S depending on where the PE was disabled; the block to which an enabled PE may be mapped is already stored in the ART. In the ART, there is a pointer to an apropriate SABL for each block in S. The VPL compiler, therefore, only produces an SABL that includes alternate instructions within the same subroutine scope as that of the conditional. This information is used to determine at what point in the SPMD program the PEs should begin execution as will be explained in Subsection 3.2. If the maximum conditional nesting depth in the static program is *Cstatic*, the maximum size of a single SABL is $O(Cstatic)$.

If the number of interruptible points in S is I_B, then the

total asymptotic space complexity for the data structures described in this subsection is $O(Cstatic \cdot I_B)$. This is because, in the worst case, an $O(Cstatic)$ SABL needs to be created for each entry in the ART.

3. SIMD to SPMD

3.1. Overview

In this section, a function H_S is presented, such that $\hat{M}_{m:\mu}(H_S(\hat{S}_{1:k})) \equiv \hat{M}_{1:\mu} \equiv \hat{S}_{1:\sigma}$ for $1 \le m \le \mu$ and $1 \le k \le \sigma$. The discussion is divided into two parts. The first part determines an $\hat{M}_{m:\mu}$ given an $\hat{S}_{1:k}$. That is, given the location of the interrupted S program, find the position m in the M program to restart the computation. The second part involves establishing a viable state for $\hat{M}_{m:\mu}$ to begin computation, given the interrupted state of $\hat{S}_{1:k}$. The next two subsections discuss these issues in turn. Although parts of the migration procedure were implemented on the mixed-mode (hybrid SIMD/MIMD) PASM prototype [8, 10] as a proof-of-concept, a full implementation is beyond the scope of this paper.

3.2. Determining $\hat{M}_{m:\mu}$ from $\hat{S}_{1:k}$

To determine the $\hat{M}_{m:\mu}$ from $\hat{S}_{1:k}$, first consider the case where the interrupted $\hat{S}_{1:k}$ is not in the scope of a poly conditional expression. This implies that all PEs will be enabled when S is interrupted. This is because any enclosing conditional expressions are mono conditional expressions, which affect all the PEs the same way (recall that mono variables have the same value on all PEs). Because, in SIMD mode, all PEs execute the same instruction at the same time, each mono conditional expression is evaluated by all PEs simultaneously. Hence, the mono conditional expressions determine which instruction all PEs will perform next, and do not disable any of the PEs. For this reason, if all PEs are enabled, when S is interrupted at some block s_r, H_S maps all SIMD PE states to the corresponding block m_r (as specified in the ART).

The other case occurs when one or more poly conditional expressions have disabled some PEs at the time of the migration. Fig. 2 shows a high-level language version of example code segments in S and M that are equivalent. To simplify the presentation, the conditional statements only involve if-then statements and not if-then-else statements. The concepts and procedures presented here, however, are applicable to if-then-else statements. The poly variable myname holds the number of the PE for $P = 4$. The arrows illustrate the mapping of S to M by H_S for each PE depending on whether S was interrupted at s_1, s_2, or s_3. As indicated, not all PEs are mapped to the same point in M. For example, if the interruption occurs as s_2, then PEs 2 and 3 map to m_2, PE 1 to m_5, and PE 0 to m_6.

At the start of an SIMD program, each PE's enable stack has a "1" at the top of the stack. Then an entry to the stack is added for each poly conditional statement. A "1" is added if the poly conditional is evaluated "true," otherwise, a

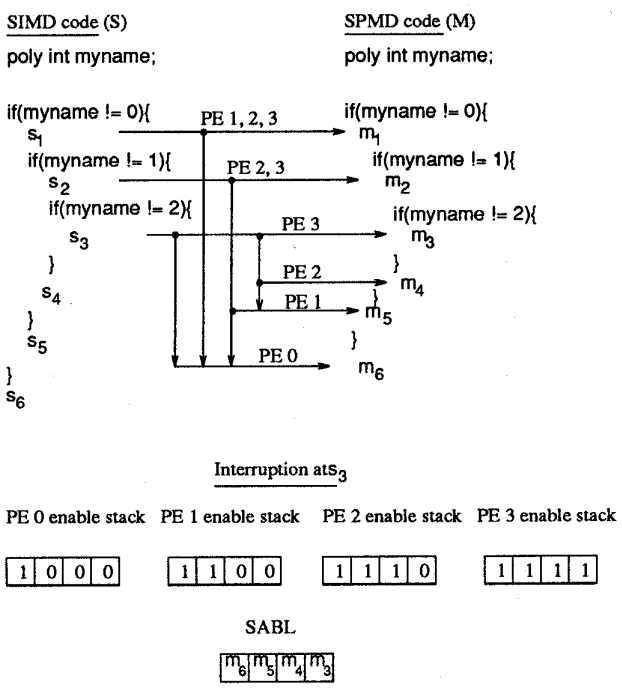

Fig. 2: Possible mappings at each interruptible point in the SIMD code, and the enable stack of each PE at the time of the interrupt.

"0" is added. The value at the top of the stack is removed when the program exits from the scope of a poly conditional. Fig. 2 also gives a graphical representation of the hardware enable stack of each PE if S is interrupted at s_3. Depending upon the enable stack, a PE can be mapped to one of four (for this example) locations in the SPMD code segment. The four locations are listed in the SABL. The SABL can be easily compared to the enable stack of the PEs. Each PE is mapped to the location in the SABL that corresponds to the last (rightmost) "1" in its enable stack. Specifically, if S is interrupted at s_3, PE 0's last "1" is in stack position 1 and thus gets mapped to location m_6. Similarly, PE 1's last "1" is in stack position 2 and thus gets mapped to location m_5, and so on. The results are applicable to other VPL statements because all other VPL statements that involve poly conditional expressions can be expressed with mono conditional expressions and poly if-then statements.

The example can be generalized for an arbitrary enable stack. This can be shown by nesting if-then statements. Consider a nesting of if-then statements of depth N in S. Fig. 3 shows a run-time nesting of these statements, which means that the if-then statements may not be nested in the static program but a PE has encountered N if-then conditionals at the time of the interrupt. That is, some of the conditional statements are not in the same subroutine. In this case, no SABL constructed at compile time will contain all the possible PE enabling locations because each SABL is based on a static nesting within a subroutine static scope. Consequently, an alternate block list (ABL) is

constructed at run-time, because some if-then statements may appear in subroutines and have different nesting depths each time the subroutine is called. If S is interrupted at code block a_N, there are $N+1$ different possible blocks to which a PE may be mapped in M. The different blocks are the corresponding block in M of a_N and the corresponding block in M of the blocks in S for which a PE should be enabled after failing one of the conditionals.

Suppose the $N+1$ blocks are ordered as in Fig. 3. Then each PE is mapped to the block in M that corresponds to the block in S at the position in the ABL that corresponds to the position of the "1" in the enable stack closest to the top of the enable stack. For example, suppose PE 3 fails the if-then statement at depth $5 < N$. The enable stack of PE 3 and the ABL at code block a_N is shown in Fig. 3. Because a_4 corresponds to the "1" in the enable stack closest to the top of the stack, PE 3 will be mapped to the block in M that corresponds to a_4.

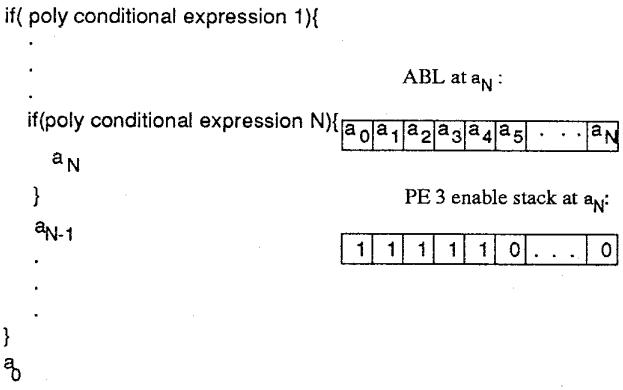

Fig. 3: Comparison of enable stack and ABL.

To form the ABL that will be compared to the PE enable stack, H_S concatenates the SABL produced by the VPL compiler for the current program counter address and each return address on the CU run-time stack. To see how this method produces the required ABL, consider the example in Fig. 4. The SABLs for different blocks of interest in the code segment are shown. Block m_i (i.e., for $i = 4,5,6$ in the SABLs) is the block in the SPMD program that corresponds to block s_i in the SIMD program. If the code segment is interrupted for task migration at the first invocation of sub(), the ABL at block s_1 is generated (as described above) by concatenating the SABL for the current block s_1 (m_6) and for the return address values on the CU's run-time stack, in this case just s_2 (m_5). If the interruption for task migration occurs during the second invocation instead, the ABL is generated from the SABL of blocks s_1 (m_6) and s_3 ($m_5 m_4$). Once the ABL is thus formed, it can be compared to the PE's enable stack to determine at which block in M to continue.

If the maximum conditional nesting depth at run-time is $\underline{c}$ and the maximum nesting depth at run-time of the subroutines is $\underline{d}$, then the worst-case asymptotic time complexity to map a PE to a location in M is $O(c+d)$. That is, the time to construct the ABL from the possibly d SABLs is $O(d)$. Then the time for each PE to compare the ABL to its enable stack is $O(c)$. Thus, the total time is $O(c+d)$.

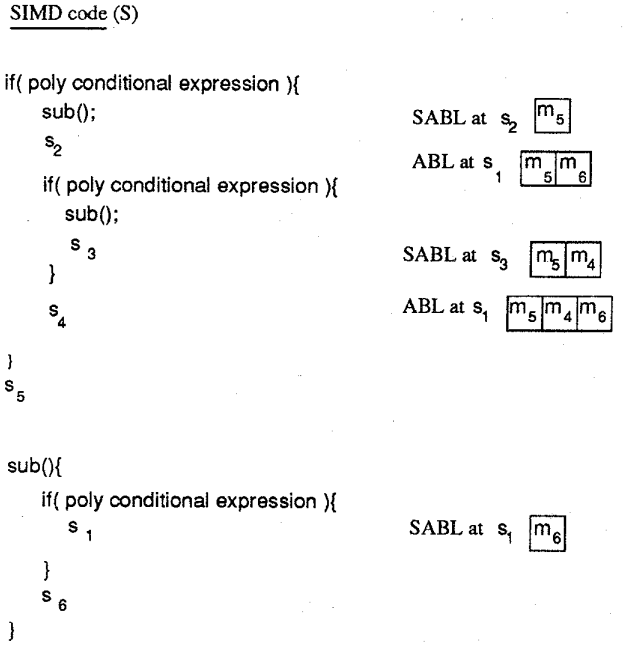

Fig. 4: An example illustrating the creation of the ABL.

3.3. Specifying H_S

3.3.1. Overview. This subsection specifies one way that H_S can map the state of an interrupted SIMD program, S, to a valid state of a SPMD program, M, where $S \equiv M$. The discussion is divided into three parts: remapping the stack, moving the data, and reallocating the registers. Although, there are other mappings that can be used for H_S, the one presented in this subsection has a time complexity that is proportional to the number of memory locations and registers that comprise the state of the program for many applications.

3.3.2. Remapping the stack. At the time of an interruption, the run-time stack of a program contains information about program subroutines that have been entered but not as yet exited (i.e., begun but not completed). It contains information about the subroutines' parameters, local variables, return addresses, and frame pointers. The way temporary values are handled is discussed later. For SPMD program, M, each PE has its own run-time stack. However, for the SIMD program, S, each PE shares its run-time stack with the CU. Because the CU executes all control flow and mono variable operations in SIMD mode, the CU stack contains all subroutine return addresses, the CU frame pointer, mono variable parameters,

and mono local variables. Each PE's stack contains the PE's frame pointer, poly variable parameters, and poly local variables.

The CU and PE stacks for a SIMD implementation and the PE stack for a SPMD implementation are shown in Fig. 5 for an example VPL function. The label rts_location is the return address of sub() and frame_ptr is the address of the previous subroutine's stack frame. It can be seen from Fig. 5 that if the PE stack and CU stack for S are combined, the PE stack for M can be created. This is done by performing the following steps: (1) copy the PE local variables, (2) copy the CU local variables, (3) remap the CU return address and regenerate the frame pointer, (4) copy the PE parameters, (5) copy the CU parameters, and (6) continue steps (1) to (5) until the stack is completely copied.

```
main(){
    poly int poly_param;
    mono int mono_param;

    sub( poly_param, mono_param );
rts_location:
    .
    .
    .
}

sub( poly_arg, mono_arg )
poly int poly_arg;
mono int mono_arg;
{
   poly int poly_local;
   mono int mono_local;

   <interrupt here>

}
```

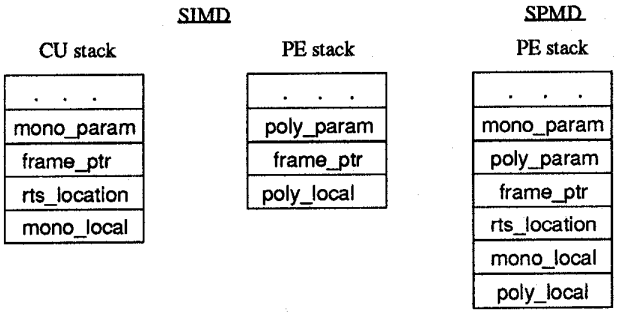

Fig. 5: A comparison of CU and PE stack contents in SIMD mode, and PE stack contents in SPMD mode.

Step (3) requires the return address and frame pointer to be remapped from the addresses on the SIMD machine to the addresses on the SPMD machine. The return addresses need to be remapped because the object code operations of M may be at different memory locations than that of S. This is done by simply using the ART (discussed in the previous section), to find the apropriate point in M that corresponds to the return location in S. Because the size of the stack in M is approximately the size of the CU stack plus the PE stack in S, the frame pointer values must be remapped. The remapping of the frame pointer is done by counting the number of memory locations copied for the mono and poly parameters of the currently scoped subroutine and the mono and poly local variables of the previously scoped subroutine, plus 1 for the return address. The SPMD frame pointers can be calculated as the SPMD PE stack is constructed from the SIMD CU and PE stacks. (The SPMD frame pointers are not a mathematical function of the SIMD CU and PE frame pointers.) The reason why the VPL disallows pointers to local variables (see Subsection 2.2) is because their contents would have to be remapped for the new stack in M. This may be prohibitively expensive, and is not considered here.

However, because each SIMD PE can be mapped to a different block in M (see Subsection 3.2), each SPMD PE may require a different stack. One possible way that H_S can determine which part of the CU stack should be merged with each SIMD PE's stack is the following. At the time of the interrupt, each SIMD PE traces the frame pointer values stored on its stack to compute the number of stack frames on its stack. Similarly, the CU uses the frame pointers stored on its stack to build a list of stack frames on its stack. Because the PE stack has not changed since it was disabled, the CU stack may have more stack frames than some PEs. However, the CU stack frames that correspond to PE stack frames are consecutive and start from the bottom of the stack. Thus, each PE uses the number of stack frames on its stack to determine the CU stack frames with which its stack frames should be merged.

Now, consider the worst-case time complexity of this procedure. H_S first moves each PE's stack from the SIMD machine to SPMD machine. The CU stack is then moved and broadcast to the PEs on the SPMD machine (assuming the interconnection network on the SPMD machine has a broadcast mechanism). Each PE on the SPMD machine merges the appropriate portion of the CU stack with its stack. Let the CU stack size be $\underline{csize}$ and PE stack size be $\underline{psize}$. The time to move the PEs' stacks is $P \cdot psize$. The broadcast of the CU stack is $O(csize)$ and the merge of the two stacks is $O(psize + csize)$. The time to determine the portion of the CU stack with which to merge the PE stack is $O(d)$, where d is the subroutine nesting depth at the time of the interrupt (as in Subsection 3.2), and $d < csize$. Thus, the total time complexity is $O(P \cdot psize + csize)$.

3.3.3. Moving the data. The movement of data from the SIMD machine, executing program S, to the SPMD machine, executing program M, is straightforward, given the assumptions of the virtual address segmentation stated in Subsection 2.2. The mono and poly variables from the SIMD machine are

copied to the corresponding mono and poly variable locations in the SPMD machine. Because the mono and poly variables share the same virtual address space in S, the variables will have the same virtual addresses in M. Thus, none of the mono variable virtual addresses will be the same as a poly variable address in M. Furthermore, pointers need not be remapped as their contents reference the same object in both S and M. Consequently, memory that may have been dynamically allocated in S can be correctly assigned by H_S and correctly referenced by M. If the number of bytes of mono and poly variable data is $\underline{Cdata}$ and $\underline{Sdata}$, respectively, then to move the data and rebuild the memory management page tables is in the worst case $O(Cdata + P \cdot Sdata)$. The $Sdata$ poly variables in all P PEs may have different values in each PE, and hence, must be moved separately.

3.3.4. Reallocating the registers. Register usage in the SIMD program, S, may differ significantly from that in the corresponding SPMD program, M. The reason for this is that on a SIMD machine, some registers exist on the CU and others on the PEs. The registers on the CU are not used for poly variable operations. On the SPMD machine, all registers are on the PEs and thus any register can be used for poly or mono variable operations. Recall from Subsection 2.2 that the SIMD machine is assumed to have $Creg$ registers on the CU and $Sreg$ registers on the PEs. Furthermore, the SPMD machine is assumed to have $Creg + Sreg$ registers on its PEs. Suppose a portion of the VPL program, F, involves many poly variable operations. Because S uses only $Sreg$ registers for poly computation, program S may have to reference memory more frequently than M for this portion of the program. Depending on the code generated for differing memory and register usage, the code in S may produce different partial results than that of the corresponding portion in M, which would affect the VPL compiler's formation of uninterruptible blocks (Subsection 2.2). The impact that this may have on the mapping function, H_S, is compiler dependent.

Consider two strategies for handling the register reallocation from a state in S to a state in M. The simpler strategy, which may result in an inefficient SPMD program, is to have M emulate the register usage in S. For example, the VPL compiler may only permit $Sreg$ registers of the $Creg + Sreg$ registers available on each PE on the SPMD machine to be used for poly variable operations. Depending on the computation, this may lead to inefficient register usage of the remaining $Creg$ registers. A similar strategy can be used for compiler generated temporary memory locations. To determine the impact of such a strategy on performance, experimental data would need to be taken for a variety of target applications.

Another strategy is to have the VPL compiler store, in the ART, temporary memory location and register mappings. A compiler keeps track of the information stored in registers and temporary memory locations as it generates and optimizes code, and could encode the relevant parts of this information in the ART. This information could then be used by H_S to map the values stored in registers and temporary memory locations at an interrupted point in S to a valid state in M, where S and M efficiently use the registers on the SIMD machine and SPMD machine, respectively. For example, each entry in the ART may point to an array that specifies a permutation of register numbers and temporary memory locations. The mapping can be done by reading this array and moving the data accordingly. If $\underline{Ctemp}$ and $\underline{Stemp}$ bytes of temporary memory data are poly and mono variables, respectively, then in the worst case the mapping would take $O(Creg + Ctemp + P \cdot (Sreg + Stemp))$ time. The particular mapping strategy and implementation depends upon the register allocation scheme used by the VPL compiler.

4. Summary

For fault-tolerance, load balancing, or various administrative reasons, a task may need to be migrated dynamically between an SIMD machine and an SPMD machine. The mapping, H_S, from the state of an interrupted SIMD virtual-machine program to a viable state in a SPMD virtual-machine program was presented. The implementation of H_S is compiler dependent, and several design trade-offs would still need to be analyzed by experimentation.

For many applications, the added burden of compiling a program to allow dynamic task migration may not be necessary and should not be done. However, when an application program must be capable of being migrated, there are time and memory space costs that are unavoidable. This paper described the necessary overhead associated with migrating a task from an SIMD virtual machine to an SPMD virtual machine.

To limit the scope of this paper it was assumed that the hardware configuration of the SIMD virtual machine and SPMD virtual machine differed only to support the different modes of parallelism. This is not the case for most existing SIMD and SPMD machine pairs. However, the goal of the paper is to solve part of the more general problem of migrating a task between two arbitrary SIMD and SPMD machines. This work is seen as a necessary step in solving this more general problem in the field of heterogeneous computing using parallel machines [9].

Acknowledgments: The authors thank Jose Fortes, Hank Dietz, Will Cohen, and Min Tan for many useful discussions about parts of this research.

References

[1] J. B. Armstrong, H. J. Siegel, W. E. Cohen, M. Tan, H. G. Dietz, and J. A. B. Fortes, "Dynamic task migration from SPMD to SIMD virtual machines," *1994 Int'l Conf. Parallel Processing*, Vol. II, Aug. 1994, pp. 160-169.

[2] H. G. Dietz and G. Krishnamurthy, "Meta-state conversion," *1993 Int'l Conf. Parallel Processing*, Vol. II, Aug. 1993, pp. 47-56.

[3] F. B. Dubach, R. M. Rutherford, and C. M. Shub, "Process-originated migration in a heterogeneous environment," *ACM Computer Science Conf.*, Feb. 1989, pp. 98-102.

[4] Y. Hollander and G. M. Silberman, "A mechanism for the migration of tasks in heterogeneous distributed processing systems," *Int'l Conf. Parallel Processing and Applications*, Sept. 1988, pp. 93-98.

[5] MasPar Computer Corporation, *MasPar MPL Manuals*, MasPar Corporation, Sunnyvale, CA, July 1993.

[6] M. A. Nichols, H. J. Siegel, and H. G. Dietz, "Data management and control-flow aspects of an SIMD/SPMD parallel language/compiler," *IEEE Trans. Parallel and Distributed Systems*, Vol. 4, No. 2, Feb. 1993, pp. 222-234.

[7] C. M. Shub, "Native code process-originated migration in a heterogeneous environment," *ACM Computer Science Conf.*, Feb. 1990, pp. 266-270.

[8] H. J. Siegel, J. B. Armstrong, and D. W. Watson, "Mapping computer-vision-related tasks onto reconfigurable parallel-processing systems," *IEEE Computer*, Vol. 25, No. 2, Feb. 1992, pp. 54-63.

[9] H. J. Siegel, J. K. Antonio, R. C. Metzger, M. Tan, and Y. A. Li, "Heterogeneous Computing," in *Handbook of Parallel and Distributed Computing*, A. Zomaya, ed., McGraw-Hill, 1995 (in press).

[10] H. J. Siegel, T. Schwederski, W. G. Nation, J. B. Armstrong, L. Wang, J. T. Kuehn, R. Gupta, M. D. Allemang, and D. G. Meyer, "The design and prototyping of the PASM reconfigurable parallel processing system," in *Parallel Computing: Paradigms and Applications*, A. Zomaya, ed., Chapman and Hall, London, UK, 1995 (in press).

[11] J. M. Smith, "A survey of process migration mechanisms," *Operating Systems Review*, Vol. 22, No. 3, July 1988, pp. 28-40.

[12] M. M. Theimer, K. A. Lantz, and D. R. Cheriton, "Preemptable remote execution facilities for the V-System," *Tenth ACM Symp. on Operating Systems Principles*, Dec. 1985, pp. 2-12.

[13] D. G. Von Bank, C. M. Shub, and R. W. Sebesta, *A Unified Model of Pointwise Equivalence of Procedural Computations*, Tech. Report TR-EE EAS-CS-93-3, Computer Science Department, University of Colorado, Apr. 1993.

[14] C. C. Weems, G. E. Weaver, and S. G. Dropsho, "Linguistic support for heterogeneous parallel processing: a survey and an approach," *Workshop on Heterogeneous Computing*, Apr. 1994, pp. 81-88.