Towards the Design of a Heterogeneous Hierarchical Machine: A Simulation Approach [1]

Zina Ben Miled†, José A. B. Fortes†, Rudolf Eigenmann† and Valerie Taylor‡

†School of ECE
Purdue University
W. Lafayette, In 47907-1285

‡Department of ECE
Northwestern University
Evanston, IL 60208-3118

Abstract

HPAM_Sim is an execution-driven simulator of heterogeneous machines. HPAM_Sim allows the simulation of target machines consisting of different processors and interconnection networks. HPAM_Sim attempts to reduce simulation time by using lightweight threads and static augmentation. Additionally, simulation can be performed either in contention or non contention mode. The results of HPAM_Sim simulations were validated using two complementary approaches. As illustrated by an example, HPAM_Sim is very flexible and allows the simulation of various heterogeneous machines with non-traditional organizations.

1 Introduction

Several analytical studies ([2], [3], [4], [5], [17] and [18]) have shown that heterogeneous architectures can provide a cost-efficient solution to high performance computing. In particular, [4] and [5] propose a hierarchy of processors and memory architecture (HPAM) as a solution for cost-effective high performance computing. HPAM (Figure 1) is a multilevel hierarchy with varying implementation for each level. The top levels of the hierarchy have a small number of fast processors. These levels can efficiently execute serial and low parallelism sections of the application. The low levels of the hierarchy have a large number of slow processors and can efficiently execute portions of the code with high degree of parallelism. HPAM is heterogeneous in several respects. Processor speed is highest for the top levels of the hierarchy and decreases with lower levels of HPAM. Memory access time and overall size increases down the levels of the HPAM hierarchy. Intra-level interconnection and communication paradigms (i.e. message passing or shared memory) also vary across levels. For ease of programming, explicit data transfer can be abstracted so that a given level can appear as a shared memory extension of the level preceding it. The distinctive features of HPAM include:

- multilevel organization
- coexistence of different networks (topology, bandwidth and latency)
- coexistence of processors with varying performance
- coexistence of different communication paradigms (message passing or shared memory)
- coexistence of memory modules with varying size and access time.

Detailed performance studies of these and other features of heterogeneous multiprocessor systems are necessary for the design and evaluation of HPAM machines. They are also necessary to validate and refine the above mentioned analytical predictions of high performance of heterogeneous multiprocessors. Unfortunately, these performance studies cannot be done with existing multiprocessor simulators which are intended for homogeneous designs (Heterogeneous simulation environments, such as Ptolemy [10] target signal processing systems). This fact provided the motivation for the development of a simulator of HPAM like machines, called HPAM_Sim, which is described in this paper.

In order to develop HPAM_Sim, previously established and tested homogeneous multiprocessor simulation techniques were evaluated in the context of providing an environment that supports the features of HPAM and targets general purpose processor based systems.

HPAM_Sim has the following characteristics:

- *fast:* HPAM_Sim combines the use of static augmentation and lightweight threads to achieve low simulation overhead.

[1] This research was supported in part by NSF grants ASC-9612133 and ASC-9612023.

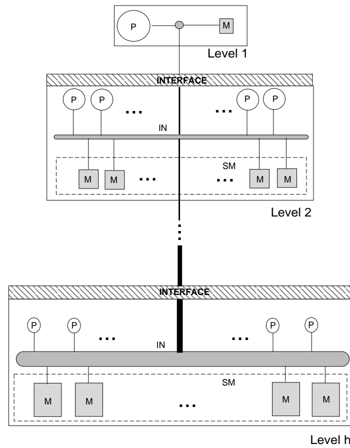

Figure 1. HPAM Organization.

- *flexible:* HPAM_Sim allows different network topologies, bandwidths and latencies to coexist. Furthermore, HPAM_Sim allows the use of different processors in the same target machine.
- *efficient:* HPAM_Sim uses resources (i.e. memory, processors and I/O) efficiently. Memory is allocated and deallocated as needed. Furthermore, it is possible to control the I/O overhead by specifying the desired level of details of the generated logfile. Finally, HPAM_Sim uses as many processors as available.

Currently the communication paradigm is restricted to message passing. Future enhancements of HPAM_Sim will allow the study of HPAM architectures with heterogeneous memory organizations and different communication paradigms.

While HPAM_Sim was mainly motivated by the need to study HPAM machines, it also provides a simulation environment for other heterogeneous as well as homogeneous architectures. Thus, HPAM_Sim can be used to compare the performance of heterogeneous machines to that of homogeneous machines. Heterogeneity is parameterized in HPAM_Sim. Therefore, it can be turned off for the simulation of homogeneous machines.

Section 2 of this paper includes a review of previous related work. A description of HPAM_Sim is provided in Section 3. Validation and experimental results are presented in Section 4 and Section 5, respectively. Conclusions and directions for future work are provided in Section 6.

2 Related Work

In general, simulation can be categorized under three major areas: model-driven, trace-driven and program-driven ([14], [8], [9], [24] and [26]). These three approaches are complementary and there are instances for which one is more appropriate than the other. The model-driven approach consists of representing the behavior of the target machine by a stochastic model [14]. This approach is concise but cannot be easily applied because an accurate model depends on detailed knowledge of the target machine. Such knowledge is not available when a machine is being developed.

Trace-driven simulation consists of generating a trace of the events under study which are then fed to a program that simulates the functional behavior of the target architecture. Trace-driven simulation is practically difficult because trace collection consumes a large amount of storage space and trace processing is time consuming [26]. Often, this technique relies on the similarities between the host and the target architecture.

Program-driven simulation consists of executing the input workload on a host machine. Each event generated by this execution is processed by the simulator [9]. This approach can be resource bound (i.e limited by the resources available on the host machine). However, it is flexible, relatively accurate, and allows the study of different aspects of the target architecture. A special case of program-driven simulation is execution-driven simulation where control is transferred to the simulator only at events of interest ([9] and [27]). The main advantage of program-driven simulation is the (feedback) ability to steer the execution of the input program according to the ordering of events dictated by the results of the simulator. Therefore, altering the behavior of the execution can be easily done. This feedback does not exist in trace-driven simulations. The generated trace is based on a certain order of execution that is difficult to modify given the information included in the trace.

In trying to analyze the features of HPAM, a model-driven approach was first considered. However, very little is known about the different parameters of HPAM to allow an accurate conception of such a model and an accurate evaluation of the needed parameters. The trace based approach was also considered from two avenues. The first avenue consists of first reorganizing existing hardware into a multilevel hierarchical setup, then using tracing tools to obtain the desired information. This avenue was not adequate because hardware reorganization is limited by existing physical hardware which allows very few features to be varied. The second avenue consists of emulating the target machine on a host machine and then using the tracing capability of the host machine to generate necessary information about the target machine. Tracing is heavily dependent on the physical features of the hardware of the host machine. Thus, it is difficult to isolate the effect of one feature of HPAM from that of another feature, a major disadvantage when designing new architectures.

There are several existing program-driven and execution-driven simulators for homogeneous multiprocessor machines. These simulators include RPPT [12], Tango-Lite

and its predecessor Tango [15], Proteus [8], CacheMire [9], WWT [20], SPASM [24], Mint [27], SimOS [22] and Augmint [23]. This list of simulators is not exhaustive. However, it is a representative set of different simulation techniques reported in the literature.

Each of the above simulators targeted one or more aspects of machine architectures. The focus of RPPT is network simulation for parallel message passing Fortran and C programs. Tango and Tango-Lite are network simulators for parallel shared memory C and Fortran programs. Tango-Lite is an enhanced version of Tango that uses lightweight threads. Proteus is a network simulator for parallel shared memory C programs. Tango, Tango-Lite and Proteus are execution-driven simulators that do not simulate local memory references. CacheMire is a distributed shared memory simulator that takes shared memory C parallel programs as input. It is a program-driven simulator that simulates both instructions and (local and shared) data references. WWT is an execution-driven network simulator for distributed shared memory multiprocessors that runs on a CM5. The main idea behind the design of WWT is to exploit architectural similarities between the target machine and the host machine. Within WWT all instructions run natively on the host except for shared memory references that generate a cache miss. Using a multiprocessor platform and the fact that only shared memory reference misses are simulated allows fast simulation of large applications. SPASM is a network simulator that takes as input parallel C message passing and shared memory programs. Reported work is only about the message passing aspect of SPASM. Mint is a program-driven simulator for shared memory parallel programs. Mint accepts any MIPS R3000 compatible object code. Therefore, Mint is not limited to a given high-level language but is restricted to MIPS based host machines. Augmint is a shared memory simulator based on Mint. However Mint is program-driven and MIPS based whereas Augmint is execution-driven and X86 based.

SimOS follows a different approach from all the above mentioned simulators. SimOS is intended to be a complete workload simulator. It cannot only simulate user level applications but also operating system calls. SimOS includes different models for each component of an architecture. The more accurate is the model the longer is the simulation time. Following an approach similar to SimOS to simulate heterogeneous systems requires a long development time. Furthermore, aside from the fact that it is operating system dependent, the level of details provided by such an approach is not necessary at the early stages of development of HPAM_Sim. At later stages of the development and prototyping of an HPAM machine, studying the effect of full workloads will be necessary and a SimOS approach should be considered and evaluated with respect to heterogeneous simulation environments.

The possibility of reusing any of the above mentioned simulators as basic building block for HPAM_Sim was considered. However, software reuse requires detailed knowledge of the inner workings of candidate simulators. This knowledge is rarely available because end-users would seldom need to modify user transparent components such as the engine of the simulator. Furthermore,in the development of previous simulators, there was no prevision of the need to introduce the kind of modifications required for a heterogeneous simulator. Reusing an existing simulator would require major recoding effort which in some cases is equivalent to the effort needed to develop a new simulator. Moreover, there are several advantages to developing a new simulator. Data structures and programming techniques that support heterogeneous simulation can be incorporated in the early stages of the simulator design process. Furthermore, the efficiency of different simulation techniques with respect to the simulation of heterogeneous machines can be taken into consideration at each step of the simulator development.

HPAM_Sim is an execution-driven, heterogeneous network simulator for message passing Fortran and C parallel programs. The choice of execution-driven simulation was based on the fact that this approach leads to a low overhead simulator and it can be easily implemented. Compared to execution-driven simulation, program-driven simulation is more accurate and detailed at the expense of longer simulation time. This is due to the fact that in program-driven simulation any event generated by the execution of the input program is simulated. In contrast, execution-driven simulation processes only events of interest. Furthermore, execution-driven simulation can adequately satisfy the requirements of an initial study of HPAM architectures.

HPAM_Sim is different from previously mentioned simulators in several aspects. Within HPAM_Sim, processors can be organized in several groups. Each group constitutes a homogeneous multiprocessor level. Furthermore, the relative processor performance of each group can be varied. Additionally, the interconnect between groups and within groups varies. These features are not supported by previously mentioned simulators and they are crucial to the study of an HPAM architecture. HPAM_Sim allows the user to easily specify and simulate architectures in which networks with different characteristics coexist. Furthermore, the user can easily specify the performance of processors in each level independently and without modifying the input code. Previously developed simulators allow users to generate this behavior by explicitly calling delay functions from within the input source code (e.g. SPASM). This feature was not intended for altering relative overall speed performance of processors. Using it for such a purpose is not only a tedious process but also error prone.

HPAM_Sim uses the feedback feature of execution-driven simulation time to slow down the execution on a

given processor or to slow down the transfer of a message on the network. Alternatively, a post processing stage can be added to account for changes in different parameters such as processor speed, network bandwidth and network latency. This alternative can be inaccurate because modifying any of the mentioned parameters can result in a change of the order of execution of certain events.

3 HPAM_Sim Organization

The input to HPAM_Sim can be either Fortran or C code with message passing directives. Assembly code corresponding to the input program is generated using a Fortran or C compiler. The generated code is the actual input to the simulator. A complete block diagram of HPAM_Sim is provided in Figure 2. Although this figure includes a shared memory simulator, a cache simulator and a scheduler, these modules are yet to be implemented (future work is discussed in Section 5).

In its current implementation, HPAM_Sim contains several modules. The details of the augmentation, engine, message passing and network modules are presented in the following subsections. The augmented standard libraries contain augmented source code that results from processing standard libraries codes through the augmentation module.

There are well established performance analysis and visualization tools which are based on extensive studies of existent systems and try to provide report summaries about potential sources of bottlenecks. HPAM_Sim takes advantage of these tools by generating a log file in PICL format [28]. This log file can be then analyzed and graphically displayed by ParaGraph [16]. There are instances where the generated log file requires a large amount of disk storage and causes high I/O overhead. For these instances, HPAM_Sim allows the user to select a concise summary of the simulation results rather than the regular log file.

3.1 Augmentation Module

The augmentation module in HPAM_Sim inserts code to account for the evolution of time during simulation. The success of the augmentation process is evaluated by its accuracy in estimating execution time and by the time overhead it adds to the simulation. The lower this overhead the more successful is the augmentation. There are two different augmentation strategies: assembly level augmentation (performed on the assembly code) and binary level augmentation (performed on the object code).

Binary augmentation is usually more accurate than assembly augmentation because it includes the exact cost of all library calls and synthetic operations. In assembly level augmentation, the libraries have to also be augmented and

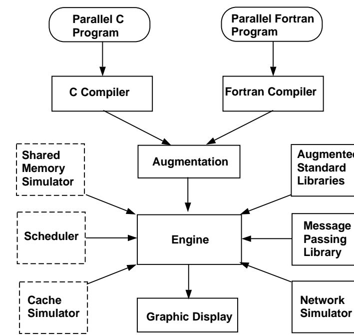

Figure 2. HPAM_Sim Organization. Solid and dashed boxes correspond to implemented and unimplemented modules respectively.

the simulator might not provide full coverage (i.e not all libraries have been augmented). TangoLite has 90% coverage [15]. The added accuracy of binary augmentation is mainly due to the fact that synthetic instructions in the assembly code are fully interpreted at the object code level. From an implementation point of view, there are two disadvantages to binary augmentation: portability and ease of implementation. Binary augmentation is machine dependent and it requires extensive knowledge about the host machine object code format. Additionally, jump targets and labels might have to be recomputed [23]. Binary augmentation is used in SimOS, CacheMire and Mint. Assembly augmentation is used in Proteus, SPASM, Augmint, RPPT, Tango and TangoLite. Augmentation in HPAM_Sim is done at the assembly level code. However, in order to reduce the inaccuracy due to synthetic operations, these operations are replaced by an equivalent assembly level routine that can be augmented. Thus, code generated by the augmentation module of HPAM_Sim does not include synthetic operations.

Augmentation can also be classified as dynamic, static or hybrid (i.e a combination of both). Dynamic augmentation is usually associated with binary augmentation. That is, at the object level each instruction is interpreted by a scheduler that models the functional units of the target processor. This results in a very accurate estimation of the execution time. However, this interpretation can be slow. In order to reduce this overhead static augmentation is used when the cost of execution is known a-priori. This hybrid approach was adopted in SimOS.

Static augmentation consists of first assigning a dedicated location in memory for cycle counting. The augmentation module then processes the input assembly code basic block by basic block and inserts code to increment the cycle count by the sum of the cost of the individual instructions in the corresponding basic block. Proteus, Tango, Tango-Lite and Augmint are examples of simulators that use static augmentation. If the purpose of the simulator is to study the detailed functionality of a given processor in a multiprocessor sys-

tem, this approach can be inadequate. However, if the intent of the simulator is to understand relative system behavior by comparing different organizations, static augmentation is sufficient. In SPASM, a different approach to static augmentation is adopted. The average cost of a basic block is estimated by collecting timing statistics over several runs of the input program. These costs are then inserted by hand into the input program via calls to a delay function. Aside from being tedious, this process is error prone [24].

In HPAM_Sim, it is assumed that an assembly instruction can be issued at each clock cycle. This assumption allows us to use static augmentation for cycle counting. However, HPAM machines have heterogeneous processors, thus some processors are faster than others. Therefore, the relative delay due to the difference in processor speed is determined dynamically (i.e at run time). In addition, within HPAM_Sim, all calls to standard library routines are replaced by calls to equivalent augmented routines. These libraries are modified versions of the BSD Berkeley distribution libraries. Given that the execution of most routines is dependent on the input operands, providing such libraries gives an accurate cycle count every time a routine is called. Additionally, it increases the portability of the simulator by making it self contained. A similar approach was used in Proteus. HPAM_Sim extends this approach also to synthetic operations by replacing each synthetic operation with an augmented assembly level routine.

As already mentioned, augmentation uses a location in memory to store the cycle counter during simulation. In the case of Proteus, Tango and Tango-Lite, this counter is an integer and each time it is incremented it is compared to a maximum value in order to prevent overflow. In HPAM_Sim an approach similar to Mint was adopted. Timing is implemented in double precision in order to allow the simulation to run for a longer period of time before transferring control to the engine which handles the overflow. This reduces the overhead of switching between the input program execution and the engine of HPAM_Sim. The disadvantage of this design decision is that it takes longer to load and update the counter. However, this overhead is balanced by the additional overhead due to checking the counter for overflow in an integer implementation.

In order to manipulate the cycle counter, a set of registers is needed to perform loads, increments and stores. The augmentation module of Proteus scans the assembly code to find unused registers and uses them in the above mentioned manipulation operations. For HPAM_Sim, a different approach was adopted. The compiler is instructed to reserve the set of registers needed by the augmentation. This choice was based on several factors. First, the compiler has highly optimized register allocation algorithms and these algorithms are improved with new releases. Therefore, HPAM_Sim can benefit from these optimized register allocation schemes. Second, the fact that the same registers are used throughout the augmentation process facilitates debugging of the augmentation module. Theoretically, the HPAM_Sim approach is intrusive because it actually eliminates a set of registers that are used for augmentation and can cause unnecessary register spills. However, practical tests using the Purdue [21], CMU [13] and Perfect-Club [6] benchmarks showed that none of the registers used in the augmentation were actually used in the assembly code generated by normal compilation. Therefore, register spills are not created by the augmentation process in the above mentioned benchmarks.

3.2 Engine Module

The purpose of the engine of the simulator is to initiate and orchestrate calls to other modules of the simulator. Furthermore, the engine is the part of the simulator that interfaces with the execution of the input program on the host machine.

The engine of HPAM_Sim is based on lightweight threads. Each virtual processor (i.e. processor of the target machine) is associated with a thread. When a processor issues a send or a receive, the engine takes control over the execution. Several simulators were based on lightweight threads, including Proteus, SPASM and Tango-lite. Lightweight threads have low overhead and can significantly help in producing fast simulators [15]. The engine of HPAM_Sim uses the Solaris multithreading library [25]. In order to increase the portability of HPAM_Sim, the engine does not rely on a specific scheduling of the threads. The engine synchronizes virtual processors based on data rather than on the order of the execution of the threads. Additionally, only basic thread subroutines that are commonly available in other thread libraries are used within the engine. These two aspects of the engine allow the substitution of the Solaris multithreading library with another library easily. Aside from the currently used Solaris multithreading library, HPAM_Sim was tested using Cthreads [19].

In addition to producing fast simulators, using lightweight threads offers an additional advantage. HPAM_Sim can be used on a uniprocessor host as well as any multiprocessor host that supports multithreading. This allows the simulation of small benchmarks on uniprocessor hosts and the simulation of large benchmarks on multiprocessor hosts. The timing results included in this paper correspond to the execution of HPAM_Sim on a uniprocessor host. However, the validation of HPAM_Sim results was performed using both a uniprocessor as well as a multiprocessor host (2-and 4-processors UltraSparc).

The engine can work in two different modes depending on whether contention in the network is simulated or not. Contention in the network occurs when two or more messages compete for the same path in the network.

Non Contention Mode:

In this mode, the engine stores incoming messages in a message buffer shared by all the virtual processors. On a receive, the engine checks whether or not the message has arrived. If the message has arrived, the transfer latency of the message is evaluated regardless of the other messages in the network. If the message has not arrived, the destination processor becomes idle until it receives the corresponding message.

Contention Mode:

In this mode, the engine orders the execution of sends and receives chronologically so to allow exact simulation of network transfer operations as explained in Section 3.4. This ordering is enforced mainly through the information contained in the message header and the evolution of time in the virtual processor. The message header contains the source, the destination, the size, the *timestamp_in* and the *timestamp_out* of the message. The *timestamp_in* corresponds to the time the message was introduced in the network and the *timestamp_out* corresponds to the time the message was removed from the network and delivered to the destination. This header is used by the network module to evaluate transfer latency. It is removed by the network module when it is no longer needed.

Each virtual processor maintains a status time and a communication time. For a given processor i, the status time (st_i) corresponds to the time accumulated through execution-driven simulation up to the current communication operation. The lookahead time (lt_i) is used to advance the simulation process when processor i is idle (i.e. waiting for a message). Additionally there are two flags associated with each processor i, $Not_Waiting_i$ and END_i. When processor i is not waiting for a message, the flag $Not_Waiting_i$ is set to *true*. When processor i terminates execution, END_i is set to *true*.

Each virtual processor is assumed to have an incoming and an outgoing queue. There is a cost associated with the transfer of a message out of the incoming queue or into the outgoing queue. Let i and j denote the source and the destination of a message (msg_{ij}) respectively. Additionally, let ϵ_{ij} denote the time needed by processor j to put msg_{ij} on its outgoing queue. On a send, msg_{ij} with its header is inserted in the message buffer in increasing order of *timestamp_in*. On a receive, if the message is not in the message buffer then there are two possible scenarios. The first scenario is when processor j is not itself waiting for a message, then msg_{ij} cannot be delivered to processor i prior to the status time of processor j (st_j) plus the time needed by processor by j to put msg_{ij} on its outgoing queue (ϵ_{ij}). Thus the *timestamp_in* of msg_{ij} cannot be less than $st_j + \epsilon_{ij}$. This limit is the amount of lookahead corresponding to processor i that is available to the engine. The second scenario is if processor j is waiting for a message. Processor j needs to receive its own message first, then it can send msg_{ij} to processor i. Therefore, msg_{ij} cannot be delivered to processor i prior to the lookahead time of processor j plus the time needed by processor j to put msg_{ij} on its outgoing queue. The lookahead time of processor i for the two scenarios can be expressed by the following equations:

if ($Not_Waiting_j$)
    $lt_i = st_j + \epsilon$
else
    $lt_i = lt_j + \epsilon$

For the two scenarios, the lookahead time is always safe and therefore no rollback is needed. This lookahead possibility reduces the overhead of switching between two virtual processors and increases concurrency in HPAM_Sim.

If on a receive, the message is in the message buffer, then the engine checks if there are no possible other messages that are chronologically ahead of msg_{ij}. This process is performed by checking if any processor k that has not terminated simulation (i.e. END_k is set to false) satisfies the following condition:

($st_k <= timestamp_in$) and
(($Not_Waiting_k$) or ($lt_k <= timestamp_in$))

In the above condition *timestamp_in* refers to the time msg_{ij} was introduced in the network. If two processors attempt to send a message at the same time, precedence is given to the processor with the smallest global number (i). This ordering is needed to guarantee a deterministic behavior of HPAM_Sim across different simulation runs on the same host machine and across different host machines regardless of the scheduling strategy of the underlying threads. It is enforced by using the above condition for $k < i$ and changing the two inequalities ($<=$) to strict inequalities ($<$) for $k > i$.

3.3 Message Passing Module

The message passing library was modeled after the MPI library. MPI code requires minor modifications in order to be ported to a valid HPAM_Sim input (if necessary the modifications could be made systematically so to be transparent to the user). This allows the use of the large amount of MPI code available to test the simulator as well as the machines to be simulated. The syntax and semantics of a non-blocking send and blocking receive in HPAM_Sim when contention is not simulated are as follows:

hpam_nsend(destination,level,type,message,size,tag)
    store message in the message buffer

```
hpam_brecv(source,level,TYPE,message,size,tag)
    if (message has arrived)
        evaluate transfer latencies
        retrieve message from the message buffer
    else
        yield to other processors
```

When contention is simulated the syntax and semantics of a non-blocking send and a blocking receive in HPAM_Sim are as follows:

```
hpam_nsend(destination,level,type,message,size,tag)
    store message by increasing time_stamp_in

hpam_brecv(source,level,TYPE,message,size,tag)
    if (message has arrived)
        if (message has smallest time_stamp_in)
            evaluate transfer latencies
            retrieve message from message buffer
        else
            yield to other processors
    else
        advance time stamp
        yield to other processors
```

In a one level multiprocessor organization each processor is identified with a unique number (node identifier). For the HPAM multilevel architecture, each processor has two identifiers. The first is the processor local number within its level. The second is the processor global number with respect to the entire architecture. Having these two identifiers allows the user to express inter-level and intra-level communication easily. For a given level of the architecture there is a direct mapping between the local number of a processor in a given level and its global number. The user needs to specify not only the source or destination local number but also the level. For example, processor 0 of level 1 can send or receive a message from processor 0 of level 2.

In addition to send and receive operations, several other message passing operations are provided in HPAM_Sim. These include arithmetic reduction, logical reduction and broadcast operations. The implementation of these operations are based on a binary tree reduction algorithm.

3.4 Network Module

The network module in HPAM_Sim is different from any implementation of the simulators introduced in the related work section. First, in HPAM_Sim, the network module has two major components, the intra-level interconnect and the inter-level interconnect. Figure 3 shows an example HPAM configuration. Each level of HPAM can have a different interconnect. Furthermore, the interconnect latency and bandwidth can vary from one level to the next. Given the latency, bandwidth and topology of the network, the network module evaluates the transfer latency for a given message. The evaluation of the transfer latency follows the model discussed in [1].

Let S, W, and L denote the size of the message, the network point to point bandwidth and point to point latency respectively. The topology of the intra-level interconnection network can be either a bus, a cross-bar or a mesh. This selection is representative of the types of networks that are used in commercial multiprocessor machines. For a given message, with no contention, the transfer latency (tl) for a bus and a cross-bar are given by the following two equations.

$$tl_{bus} = L * (\frac{S}{W}) \quad (1)$$

$$tl_{cross-bar} = L * (\frac{S}{W}) \quad (2)$$

The routing for the mesh topology is a dimension ordered x-y routing. Furthermore, the mesh network is assumed to be circuit-switched and pipelined. This mesh network model can be easily extended to other configurations (e.g wormhole routing, packet-switched) when needed. Given that the number of hops from the source to the destination is n, the transfer latency with no contention for the mesh topology is given by the following equation.

$$tl_{mesh} = L * (\frac{S}{W} + n) \quad (3)$$

The inter-level interconnect is based on a point to point connection between two nodes from two different levels. Therefore, the inter-level transfer latency between directly connected processors is similar to the transfer latency between two nodes in a cross-bar network.

As previously mentioned, the network module routes messages in chronological order when the network contention is simulated. In this mode and in order to exactly evaluate contention, the header of all the messages remain in the message buffer until it no longer needed. Let msg be the message being routed by the network module. The message buffer is examined and any message that has a *timestamp_out* greater than the *timestamp_in* of msg is discarded. Given the remainder of the messages, the delay due to the the transfer of msg is evaluated. This delay is the sum of the transfer latency with no contention and the delay due to contention. As explained in the next paragraph, the contention and its corresponding delay depends on the topology of the network. When the network contention is not simulated the message and its header are removed as soon as they are retrieved by the destination processor.

For a bus based interconnect, only one message can be on the bus at a time. Therefore, each message has to wait for all pending messages to clear from the bus. The cross-bar interconnect has a dedicated point to point connection

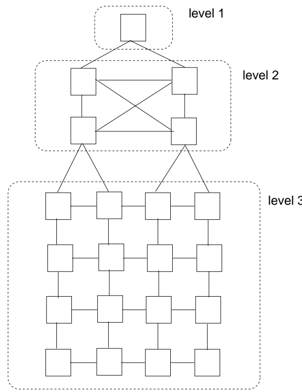

Figure 3. Three-Level HPAM Architecture Example.

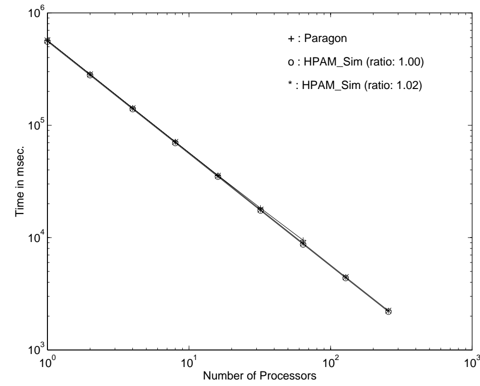

Figure 4. HPAM_Sim Validation for Problem01.

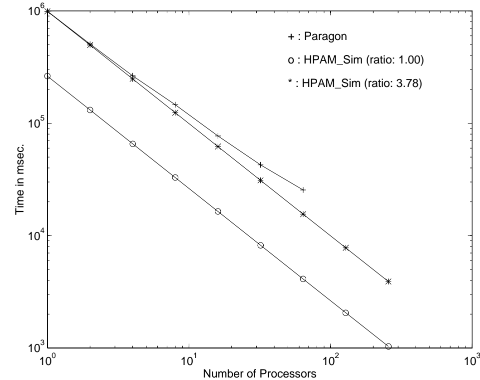

Figure 5. HPAM_Sim Validation for Problem04.

between any two nodes. Within HPAM_Sim contention between a current message and a previous message in a cross-bar can occur only if the two nodes involved in the two messages are the same. Given the assumed routing mechanism for the mesh topology, two messages collide in the mesh network if and only if their paths cross. In the event of a collision, the message that last entered the network waits for its path to clear.

4 Validation

There are two approaches to the validation of a simulator that have been reported in the literature. The first approach consists of comparing the results of a previously tested and established simulator to the newly developed simulator. The second approach consists of comparing the result of the newly developed simulator to the results of an actual execution on an existing machine. For example, the first approach was used in the validation of WWT and the second approach was used in the validation of Proteus.

Given the unique hierarchical organization of HPAM, the validation of HPAM_Sim had to rely on both of the above two mentioned validation approaches as explained in the following two subsections.

4.1 Validation of the Augmentation Module

The performance of the augmentation module is measured by its correctness, accuracy and overhead. The correctness of the augmentation establishes whether or not (under the assumed conditions) the generated result is correct. Given that in HPAM_Sim it is assumed that an instruction is issued every cycle, it is possible to verify that the results generated by the augmentation module of HPAM_Sim are correct using Spixtools [11]. This verification was performed by first using Spixtools to evaluate the total number of cycles needed for the execution of the non-augmented sequential code. This number was then compared to the total number of cycles obtained from running the same code on one processor using HPAM_Sim. The results of Spixtools and HPAM_Sim match for all of the fifteen Purdue benchmarks. This test establishes the correctness of the augmentation module.

Evaluating the accuracy of augmentation consists of comparing the number of cycles obtained from the execution of the non augmented application on the host machine (native execution) and comparing it to the number of cycles obtained from HPAM_Sim when the target machine is the same as the host machine. This test was performed on the Purdue benchmark suite using a Sun Workstation (85Mhz Sun Microsparc). Ideally, the number of cycles obtained through native execution and HPAM_Sim would be the same. However, in practice it is application specific and in this experiment the ratio of the two numbers varied from 0.33 to 0.81. Given that native execution and HPAM_Sim are operating on identical input codes, the differences in the number of cycles are due to two factors. The first factor is that HPAM_Sim does not take into account delays due to memory reference misses. The second factor is that HPAM_Sim assumes that

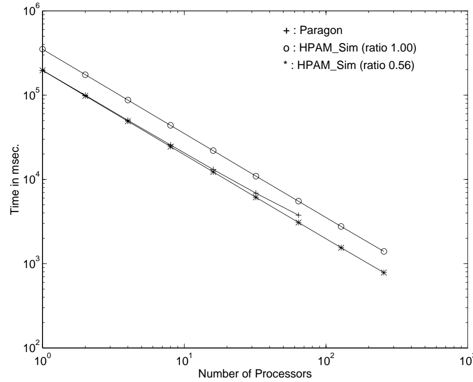

Figure 6. HPAM_Sim Validation for Problem07.

an instruction can be issued at every clock cycle. However, it might not be possible to sustain this rate due to data dependencies and resource limitations. A cache simulator and a resource scheduler can be used to correct for the inaccuracies due to the first and second factor respectively. Future work will address the development of these two components. However, even without these components which affect local processor operations, it is still possible to study the relative behavior of different applications on different HPAM architectures as shown in the next subsection.

The overhead of augmentation is the time contributed by the augmentation to the total execution time of the augmented application. This overhead is obtained by comparing the execution time of the non-augmented code to the execution time of the corresponding augmented code. This experiment showed that the ratio of the two execution times varied from 1.1 to 3.7. It is possible to improve this ratio by trying to combine basic blocks during augmentation. This optimization is used in Proteus. However, after measuring the overhead introduced by the augmentation in HPAM_Sim and comparing it to the overhead reported in the literature after optimization [8], it was deemed unnecessary to introduce this optimization because of its low return.

4.2 Validation of the Network Module

In this validation experiment HPAM_Sim was used to simulate an architecture similar to the Intel Paragon XP/S. The Paragon is a homogeneous mesh-connected multiprocessor. This configuration corresponds to a one-level HPAM with mesh intra-level interconnect. The execution of the Purdue benchmarks on the Paragon was compared to the results obtained from HPAM_Sim. The results corresponding to five of these benchmarks when the network contention is simulated are included in this section. Table 1 summarizes the important characteristics of these benchmarks.

Problem01 of the Purdue suite evaluates a trapezoidal rule estimate of an integral. Problem04 computes the sum

Benchmark	Communication Pattern	Data Set Size (KB)	Mathematical Functions Called
Problem01	regular	< 1	exp
Problem04	regular	4096	none
Problem07	irregular	512	none
Problem10	irregular	40	sin
Problem13	regular	4096	log,sin and cos

Table 1. Purdue benchmarks characteristics.

of the terms of a harmonic series. Problem07 uses Lagrange interpolation to compute polynomial interpolant values of a function at five points. Problem10 computes the LU factorization of a matrix using Gauss elimination with pivoting. Finally, Problem13 computes the sum of the terms of a power series. This benchmark selection is intended to cover different possible simulation scenarios (i.e regular and irregular communication patterns, small, medium and large data set sizes). Furthermore, this selection includes benchmarks that contain mathematical library routine calls. HPAM_Sim is intended for scientific applications as well as general purpose applications. Therefore, it is important to include mathematical as well as non-mathematical benchmarks.

In this experiment the codes executed on HPAM_Sim and the codes executed on the Paragon are different. The first uses HPAM_Sim message passing routines, while the second uses the MPI library on the Paragon. Furthermore, the compilers used to compile the two source codes are different. Finally, standard library routines called by the two source codes are also different.

Figures 4 through 8 summarize the result of this validation experiment. In each of these figures there are three plots. The first corresponds to the timings obtained on the Paragon for different number of processors. The second and third plots are obtained from HPAM_Sim simulations. In the second plot, the processor number of cycles per seconds was set to that of the building processor of the Paragon. In the third plot, this parameter was adjusted by a factor to account for delays due to memory accesses, data dependencies and instruction scheduling conflicts. This factor is application dependent. It was estimated as the ratio of the execution time of the benchmark on one processor of the Paragon to the execution time of the same benchmark on one processor with the number of cycles per second equal to that of a Paragon processor using HPAM_Sim.

One might expect this ratio to be always greater than 1.0 because it represents how far from ideal is the actual execution. In the case of Problem07 and Problem10, this ratio is less than 1.0 (0.56 and 0.61 respectively). This result can be explained by the fact that two different compilers and two

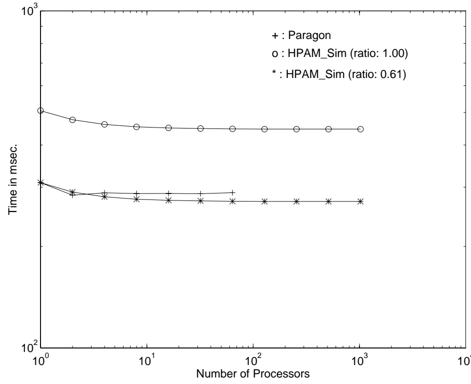

Figure 7. HPAM_Sim Validation for Problem10.

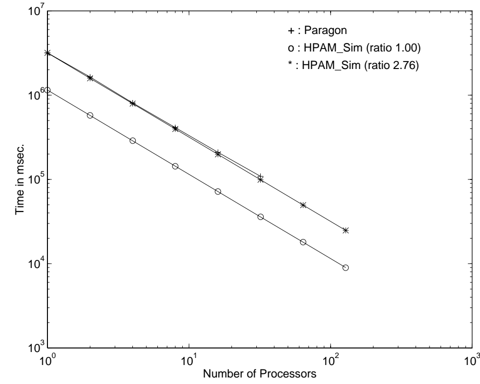

Figure 8. HPAM_Sim Validation for Problem13.

different libraries were used in the two cases. Furthermore, the instruction set for the two processors is different. In [8] and references there in, it was found that two different compilers can lead to as much as 33% difference in simulation results.

The result of this experiment (Figures 4 through 8) show that HPAM_Sim timings closely follow real timing obtained on the Paragon for all five example benchmarks. In these figures, the difference between the Paragon and HPAM_Sim timings increases as the number of processor increases. This increase in difference is mainly due to the fact that HPAM_Sim does not account for the time it takes to route the message header. The size of the message header in the Paragon is 48 Bytes. This difference becomes more apparent as the number of messages in the application increases. For all the benchmarks presented in this paper, the number of messages increases as the number of processors increases.

5 Experiment

The three-level HPAM architecture shown in Figure 3 was simulated using HPAM_Sim. The input program used for this experiment is Problem07 of the Purdue suite. Different sections of this benchmark were assigned to different levels of the architecture. Figure 9 shows the execution time (y-axis) for this benchmark against different ratios (x-axis). These ratios represent how much slower one level is in comparison with the preceding level. For example, if the ratio is equal to two, then any processor in level three is twice as slow as any processor in level two. Similarly, any processor of level two is twice as slow as any processor of level one.

Figure 9 shows three different plots. Each of these plots corresponds to a different number of processors in level three (i.e 8, 16, 32). The second configuration is the one depicted in Figure 3. The intent of this example is to show the types of HPAM organization that can be simulated by HPAM_Sim. Qualitative studies of the results are beyond the scope of this paper.

6 Conclusions and Future Work

This paper presents an execution-driven simulator for heterogeneous machines. The development of HPAM_Sim was motivated by the need to study a proposed heterogeneous multilevel architecture (HPAM).

HPAM_Sim extends simulation techniques previously used in several homogeneous multiprocessor simulators in order to provide a testbed for the study of heterogeneous multiprocessor machines. Heterogeneous multiprocessor machines are in general more complex than homogeneous machines. However, they are often more cost-efficient than homogeneous machines. HPAM_Sim provides an environment that can be used to validate this claim. Furthermore HPAM_Sim will allow a better understanding of the effect of varying the number of processors, processor speeds and the interconnection network at each level of HPAM.

There are no commercially available machines with an architecture organization similar to HPAM. Furthermore, there are no available tools that can simulate heterogeneous machines similar to HPAM. Therefore, in order to validate the results generated by HPAM_Sim a piecewise approach had to be adopted. First, the results of the augmentation module of HPAM_Sim were validated using Spixtools. Second, the results of an overall simulation of a one-level homogeneous HPAM were validated with actual runs on an Intel Paragon. Finally, the functionality of HPAM_Sim was validated for a multilevel HPAM configuration.

The development of HPAM_Sim is an evolving project which is mainly driven by the study of the HPAM architecture. In its current stage of development, HPAM_Sim allows simulation of C and Fortran message passing parallel programs. Future enhancements include most importantly the implementation of the shared memory component of HPAM_Sim. This additional module is needed to allow the analysis of heterogeneity with respect to memory organization and communication paradigm in an HPAM architecture. There are several tools, such as ADAPTOR [7], that provide

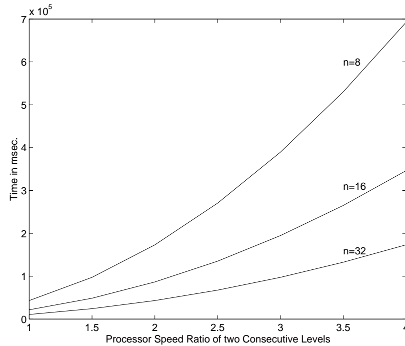

Figure 9. Simulation of a three-level HPAM.

a shared memory interface to message passing platforms. The possibility of using an approach similar to ADAPTOR will be investigated.

Although HPAM_Sim was primarily intended as a tool for the study of HPAM architectures, it can be used for the simulation of various other homogeneous and heterogeneous architectures.

References

[1] A. Agrawal. Limits on Interconnection Network Performance. *IEEE trans. Par. and Dist. Systems*, 2(4):398–412, October 1991.

[2] Andrews, J.B. and Polychronopoulos, C.D. An Analytical Approach to Performance/Cost Modeling of Parallel Computers. *J. Par. Dist. Computing*, 12(4):343–356, August 1991.

[3] Barton, M.L. and Whiters, G.R. Computing Performance as a Function of the Speed, Quantity, and Cost of the Processors. *Supercomputing*, pages 759–764, 1989.

[4] Ben-Miled, Z., Eigenmann, R., Fortes, J.A.B., and Taylor, V. Hierarchical Processors-and-Memory Architecture for High Performance Computing. *Frontiers of Massively Parallel Computation Symp.*, pages 355–362, October 1996.

[5] Ben-Miled, Z. and Fortes, J.A.B. A Heterogeneous Hierarchical Solution to Cost-efficient High Performance Computing. *Par. and Dist. Processing Symp.*, pages 138–145, October 1996.

[6] Berry, M., Chen, D., and al. *The Perfect Club Benchmarks: Effective Performance Evaluation on Supercomputers*. CSRD, Univ. of Illinois, Tech. Rep. UIUC-CSRD-827, July 1994.

[7] Brandes, T. and Zimmermann, F. . ADAPTOR - A Transformation Tool for HPF Programs. *Programming Environments for Massively Parallel Distributed Systems, Birkhauser Verlag*, pages 91–96, 1994.

[8] Brewer, E., Dellarocas, C.N, Colbrook, A., and Weihl, W.E. *Proteus: A High-Performance Parallel-Architecture Simulator*. LCS, Massachussetts Institute of Technology, Tech. Rep. MIT-LCS-TR-516, 1991.

[9] Brorsson, M., Dahlgren, F., Nilsson, H., and Stenstrom, P. The CacheMire Test Bench - A Flexible and Effective Approach for Simulation of Multiprocessors. *Simulation Symp.*, pages 41–49, 1993.

[10] Chang, W.T., Ha, S., and Lee, E.A. Heterogeneous Simulation - Mixing Discrete-Event Models with Dataflow. *VLSI Signal Processing*, 1996.

[11] Cmelik, R.F. and Keppel, D. *Shade: A Fast Instruction-Set Simulator for Execution Profiling*. Sun Microsystems, Inc., Tech. Rep. TR-93-12, July 1993.

[12] Convington, R.G., Madala, S., Mehta, V., and Jump, J.R. The Rice Parallel Processing Testbed. *Measurement and Modeling of Computer Systems Conf.*, pages 4–11, 1988.

[13] Dinda, P.A., Gross, T., and al. *The CMU Task Parallel Program Suite*. School of Computer Science, Carnegie Mellon Univ., Tech. Rep. CMU-CS-94-131, January 1994.

[14] D. Ferrari. *Computer Systems Performance Evaluation*. Prentice Hall, 1978.

[15] Goldschmidt, S.R. and Davis, H. *Tango Introduction and Tutorial*. CSL, Stanford Univ., Tech. Rep. CSL-TR-90-1410, January 1990.

[16] M. Heath. Recent Developments and Case Studies in Performance Visualization using ParaGraph. *Performance Measurement and Visualization of Parallel Systems, Elsevier Science Publishers*, pages 175–200, 1993.

[17] Menasce, D. and Almeida, V. Cost-performance Analysis of Heterogeneity in Supercomputer Architectures. *Proc. Supercomputing Conf.*, pages 169–177, November 1990.

[18] Moncrieff, D., Overill, R.E, and al. Heterogeneous Computing Machines and Amdahl's Law. *Parallel Computing*, 22(3):407–413, 1996.

[19] Mukherjee, B., Eisenhauer, G., and Ghosh, K. A Machine Independent Interface for Lightweight Threads. *Operating Systems Review of the ACM Special Interest Group in Operating Systems*, pages 33–47, January 1994.

[20] Reinhardt, S.K., Hill, M.D., Larus, J.R., Lebeck, A.R., Lewis, J.C., and Wood, D. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. *Measurement and Modeling Conf.*, pages 48–60, May 1993.

[21] Rice, J.R. and Jing, J. *Problems to Test Parallel and Vector Languages - II*. Computer Science Department, Purdue University, Tech. Rep. CSD-TR-1016, September 1990.

[22] Rosenblum, M., Herrod, S.A., Witchel, E., and Gupta, A. Complete Computer Simulation: The SimOS Approach. *Parallel and Distributed Technology*, Fall 1995.

[23] Sharma, A. and al. *Augmint: A multiprocessor Simulation Environment for Intel X86 Architectures*. CSRD, University of Illinois, Tech. Rep. CSRD-1463, December 1995.

[24] Sivasubramaniam, A., Singla, A., Ramachandran, U., and Venkateswaran, H. A Simulation-Based Scalability Study of Parallel Systems. *Par. and Dist. Computing J.*, 22(3):411–426, September 1994.

[25] Sunsoft. *Solaris Multithreaded Programming Guide*. Sunsoft, 1995.

[26] Uhlig, R.A. and Mudge. T.N. Trace-driven Memory Simulation: A Survey. *Computing Surveys*, submitted for publication.

[27] Veenstra, J.E. and Fowler, R.J. *Mint Tutorial and User Manual*. Computer Science Department, University of Rochester, Tech. Rep. 452, August 1994.

[28] P. Worley. *A New PICL Trace File Format*. Oak Ridge National Laboratory, Tech. Rep. TM-12125, September 1990.