bold, 16pt)

Hierarchical Processors-and-Memory Architecture for High Performance Computing [1]

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

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

[‡]Department of EECS
Northwestern University
Evanston, IL 60208-3118

Abstract

This study outlines a cost-effective multiprocessor architecture that takes into consideration the importance of hardware and software costs as well as delivered performance in the context of real applications. The proposed architecture (HPAM) is organized as a hierarchy of processors-and-memory (PAM) subsystems. Each PAM contains one or more processors and one or more memory modules. The following factors drive the HPAM design

application behavior – an important application behavior (locality of parallelism) is characterized and quantified for a set of benchmarks; two classes of applications that demand 100 TOPS computation rates are also characterized.

cost-efficiency – a favorable comparative analysis of a 2 level HPAM and a conventional multiprocessor is done using empirical data; technology trends that support the desirability and viability of HPAM organizations are also discussed.

ease-of-usage – a flexible programming environment for HPAM is proposed; the scenarios addressed include automatic translation systems, library based programming and performance-guided coding by expert programmers.

1 Introduction

Important computer applications have been identified that will require at least 100 Teraops (10^{14} operations per second) computing speeds [St et. al 95]. A machine that aims at providing such level of performance will require orders of magnitude more resources than current high-performance computers. As a consequence, cost-effectiveness and programmability are the overriding design issues for such a machine. The heterogeneous multiprocessor system discussed in this paper is organized as a hierarchy of processors-and-memory subsystems (HPAM). Our approach is to leverage as much as possible features of commodity microprocessors (current and expected in the future). In particular, the proposed approach exploits the analogy of the HPAM organization with conventional memory hierarchies, which microprocessors readily support.

The HPAM approach meets the spirit of the following five lessons, learned from previous multiprocessor research and development, in a *cost-effective* manner.

- the cost of multiprocessors can be greatly reduced by reusing standard commodity parts and software; cost-performance analysis must reflect this reality. //
- high software development costs deter potential customers of parallel processors; successful multiprocessor designs should present users with familiar programming environments and allow (user and system) software reuse.
- designers must always be aware of Amdahl's law; any serialization introduced in the system severely limits multiprocessor speedups.
- memory access and low level communication latencies are fundamental limitations; while bandwidths can always be increased (in theory), latencies result from fundamental limits of physics and communication software overhead, and thus are to be avoided or hidden; in practice, costs can limit bandwidth which, in turn, may increase latencies.
- real applications with irregular control and data structures should be included in the suite of benchmarks used to evaluate multiprocessor designs.

We expand on the HPAM architecture in the following section. Issues related to applications behavior, cost and performance are addressed in Sections 3 and 4 respectively. The programming environment of HPAM is described in Section 5. Section 6 includes future work and concluding remarks.

2 HPAM Architecture, Organization and Rationale

Figure 1 shows a high level view of an HPAM. An HPAM consists of several levels of processors-and-memory (PAM) systems. A PAM system contains one

[1]This research was partially funded by the National Science Foundation xxx.

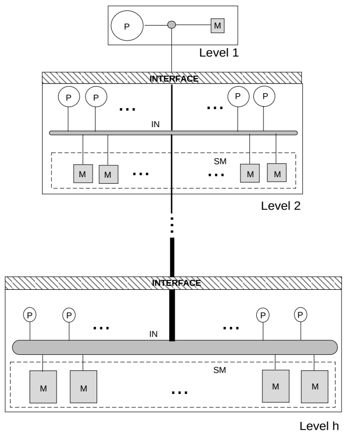

Figure 1: HPAM Organization

or more processors and one or more memory modules. Each PAM system can be implemented with different processors and interconnections at different levels. The following advantages are inherent to this kind of organization:

- it preserves the advantages of a memory hierarchy (i.e., top levels include only a "small" amount of expensive fast memory but perceive an inexpensive "very large" fast memory) and leverages of very efficient support mechanisms already available in commodity processor, interface and memory parts.
- it reduces the number of memory accesses and memory access time by computing-in-memory in a given level.
- it minimizes Amdahl's law impact by providing fast processors to execute code sections with low parallelism.
- it can adapt to variable irregular parallelism by dynamically moving data and code execution across levels (just as memory hierarchies respond to data misses).

From a programming perspective, it is desirable to have each PAM appear as a shared memory multiprocessor. However, the actual implementation may result from either a software or hardware emulation of shared memory over physically distributed memory. For example, a PAM could be a single bus machine with a physically shared memory, a symmetric multiprocessor, a cache coherent (CC) NUMA multiprocessor, a cache only machine (COMA), a non-cache coherent (NCC) NUMA machine or a massively parallel message-passing machine with distributed memory. The particular implementation of a given level would depend on the number of processors. The above example roughly orders implementations according to the increasing number of processors that they would efficiently support in the context of a shared memory programming environment (i.e. the lower the level the larger the number of processors and the harder it is to efficiently implement shared memory). However, even when memory inside a PAM appears to be distributed among its processors (as in the case of the lowest levels of the hierarchy with very large numbers of processors), the PAM appears as a single address space to the other PAMs in the hierarchy.

The use of different architectures at distinct levels of HPAM is one of the heterogeneous aspects of the proposed design. Heterogeneity is also present in the fact that processors may differ in performance and microarchitecture. The top HPAM level consists of a very fast uniprocessor and associated fast memory. Although the second level of an HPAM is actually a shared memory multiprocessor, the interface between the first and second HPAM levels is such that the second HPAM level can appear as the next level of memory for the uniprocessor for the non-knowledgeable programmer. This abstraction can be bypassed by knowledgeable programmers as discussed in Section 5.

The implementation of the second HPAM level can take one of the forms mentioned above depending on the number of processors. The third HPAM level can appear as the next memory level for the second HPAM level and is programmed as another shared memory multiprocessor system and so on. In other words, each PAM level is viewed as a shared-memory multiprocessor "wrapped" (possibly with hardware support) to look as memory to the level that precedes it.

3 Applications Behavior

Understanding the behavior of target applications is a critical aspect of architecture design. Furthermore, when seeking a multiprocessor design for a large range of applications it is imperative to consider applications whose parallelism profile varies over time. In fact, most applications will have periods of sequential, low-, medium- and high-parallelism. When taken in isolation, each of these application segments runs best on a number of processors that "matches" the parallelism available. Furthermore, fast (thus expensive) processors should be used for the least parallel application segments in order to reduce the impact of Amdahl's law. Using processors of different speeds follows the spirit of this rule while trying to achieve a low cost. Two-level architectures that explore this fact were proposed and analyzed in [AnPo 91, MeAl 90], the conclusion being that cost-effective performance improvements resulted from such an approach. A very recent study [Mo et al. 96], explores the effect of Amdahl's law on heterogeneous machines and concludes that heterogeneous machines can deliver higher theoretical performance than homogeneous machines. These and our own studies provide some of the rationale for HPAM to include several multiprocessor levels, each with distinct number and type of processors.

A perhaps conjecturable but less obvious characteristic of applications is the fact that *parallelism* has temporal locality for both instructions and data. Our empirical studies reveal that this behavior leads to little communication being required between HPAM levels when they execute different parts of an application.

The following two subsections discuss issues related to locality of programs with respect to the degree of parallelism and constraints imposed by petaflops applications.

3.1 Preliminary Findings

Locality of parallelism has been characterized and empirically studied in [BeFo 96]. An abbreviated discussion of this study follows. For this initial experiment, the hierarchy is restricted to two levels only. The first level consisted of a single fast processor. Three cases were considered for the second level (namely, 10, 100 and 1000 processors) in order to gain some insight into the locality behavior of different degrees of parallelism which might be present in multilevel machines.

For each of several benchmarks (discussed later in this section), the experiment consists of executing a program (on its associated benchmark data) on a uniprocessor. The program consists of a sequence of assembly level instructions with execution control directives. A leader is one of the following control directives { *begin*, *end*, *doall*, *enddoall* }. The leaders *begin* and *end* represent the beginning and the end of the program and are therefore unique. The leader *doall(k)* represents the start of a loop *do I=a,b* such that $k = b - a + 1$ and the loop is parallelizable. The leader *enddoall* represent the end of a parallelizable *do* loop.

A block b_i is the sequence of code between two consecutive leaders. That is

$$b_i =]L_i, L_{i+1}[\text{ where } L_i \text{ and } L_{i+1} \text{ are leaders} \quad (1)$$

Furthermore let the parallelism *bp* and the size *bs* of a block b_i be defined as follows :

$$bp(b_i) = \begin{cases} 1 & \text{if } L_i = \textit{'begin'} \\ k \times bp(b_{i-1}) & \text{if } L_i = \textit{'doall(k)'} \\ bp(b_{i-2}) & \text{if } L_i = \textit{'enddoall'} \end{cases} \quad (2)$$

$bs(b_i)$: the number of assembly level instructions executed between L_i and L_{i+1}.

The following example illustrates the definition of block parallelism and block size.

```
!begin
        ...

!doall(MK) do I= 1, MK

        lduw [%fp + 0x48], %l0
        lduw [%l0 + 0], %l0
        stw   %l0, [%fp - 0x10]
        or    %g0, 1, %l1
        sethi %hi(0x45c00), %l0

!enddoall
        ...
!end
```

Let b_x be the block enclosed between *doall* and *enddoall*. For this example, $bp(b_x) = MK$ and $bs(b_x) = 5$. The block parallelism does not have to be known statically and can be evaluated at runtime.

Throughout the remainder of this paper, the term *instruction* is used to refer to assembly level code instructions. Furthermore, all instructions are assumed to execute in the same amount of time. Under this assumption, the execution of a given program Pr is represented by the ordered sequence of blocks $[b_1, \cdots, b_n]$ such that $L_1 = \textit{'begin'}$ and $L_{n+1} = \textit{'end'}$.

The block parallelism reflects the *application degree of parallelism* at different instances of the execution of the application undependently of the machine on which the application is executed. For the machine degree of parallelism, let MDP be defined as follows :

MDP (*machine degree of parallelism*): the maximum number of assembly level instructions that can execute concurrently in the second level.

With respect to a given MDP, a *scalar window* is defined as follows:

$$W^s_{ij} = [b_i, \cdots, b_j] \mid \begin{cases} bp(b_{i-1}) \geq MDP & \text{or } i = 1 \text{ ,} \\ bp(b_{j+1}) \geq MDP & \text{or } j = n \text{ and} \\ bp(b_k) < MPD \text{ for } i \leq k \leq j & \end{cases} \quad (3)$$

Similarly, a *parallel window* is defined as:

$$W^p_{ij} = [b_i, \cdots, b_j] \mid \begin{cases} bp(b_{i-1}) < MDP & \text{or } i = 1 \text{ ,} \\ bp(b_{j+1}) < MDP & \text{or } j = n \text{ and} \\ bp(b_k) \geq MPD \text{ for } i \leq k \leq j & \end{cases} \quad (4)$$

Let Ep be an example program specified by the sequence $[b_1, \cdots, b_4]$ such that ($bp(b_1) = 1$, $bp(b_2) = 2$, $bp(b_3) = 10$, $bp(b_4) = 100$) and ($bs(b_1) = 30$, $bs(b_2) = 800$, $bs(b_3) = 10$, $bs(b_4) = 20$). For $MDP = 10$, this example has two windows: W^s_{12} and W^p_{34}.

The size of windows is defined as:

$$ws(W^x_{ij}) = \begin{cases} \sum_{k=i}^{j} bs(b_k) & \text{if } x = s \\ \sum_{k=i}^{j} \lfloor \frac{bs(b_k)}{bp(b_k)} \rfloor \times \lfloor \frac{bp(b_k)}{MPD} \rfloor & \text{if } x = p \end{cases} \quad (5)$$

The size of the window represents the amount of work done on each processor. For a scalar window, this amount is equivalent to the sum of all block sizes in the window. However, for parallel windows this work is distributed among more than one processor. The first term in the second summation of Equation 5 represents the amount of work done per processor given unlimited resources. For the example introduced above $ws(W^s_{12}) = 50$ and $ws(W^p_{34}) = 80$

For the program Pr and a given size h, the *percent scalar (parallel) execution time* with respect to the window size is defined as ratio of the sum of the sizes of scalar (parallel) windows with window size = h to the sum of the sizes of all scalar (parallel) windows.

As expressed in the following principles, the are two interesting types of locality with respect to the degree of parallelism:

- *Principle 1*: if a data item is referenced within a scalar (parallel) window of the program, it tends to be referenced again in the near future within a scalar (parallel) window (*data temporal locality with respect to the degree of parallelism*).

- *Principle 2*: if an instruction being executed belongs to a scalar (parallel) window, the instructions executed in the near future tend also to belong to a scalar (parallel) scalar window (*instruction temporal locality with respect to the degree of parallelism*).

We conducted an experiment to quantify and analyze parallelism locality in four benchmarks from the CMU-benchmark suite (Airshed, fft2, Radar and Stereo [Di et al. 94] and five benchmarks from the Perfect-Club suite (TRFD, FLO52, ARC2D, OCEAN and MDG [Be et al. 94]).

For each program, parallelism was first detected by Polaris [Bl et al. 94] and the resulting programs were instrumented to output the size of each window. Each window was then classified as scalar or parallel for values of MDP of 10, 100 and 1000.

Figures 2 and 3 show the histogram of accumulated percent scalar and parallel execution times with respect to scalar and parallel window sizes respectively, for the benchmark AR2CD. These scalar and parallel percentages indicate to what extent instruction temporal locality is present in a given application. If high percentages correspond to small windows then the application exhibits poor instruction temporal locality. On the other hand, if high percentages correspond to large windows than the application exhibits high instruction temporal locality. Table 1 shows window sizes which accumulated the highest percent scalar execution time and the highest percent parallel execution time for all the benchmarks. For each benchmark in Table 1, the first row indicates the highest percent scalar or parallel execution time and the second row is the size of the corresponding window. The results of this table show that most benchmarks exhibit high locality (e.g., for MDP=100, 99.7% of fft2 scalar execution time corresponds to window of size 10^7 and 84.9% of its parallel execution time corresponds to a window of size 10^5). The exceptions are benchmarks Airshed and MDG which exhibit relatively poor locality.

For each of the benchmarks used previously, data-reference traces with respect to MDP were collected. An M-hit ratio (for mode hit) and M-miss ratio are defined as follows:

- *M-hit ratio*: fraction of the total number of scalar (parallel) data references in a given scalar (parallel) window for which their last reference was also in a scalar (parallel) window.

- *M-miss ratio*: fraction of the total number of scalar (parallel) data references in a given scalar (parallel) window for which their last reference was in a parallel (scalar) window.

The trace collection in Table 2 was done over a large buffer (1,000,000 locations) in order to reduce

Bench– marks	MDP = 10		MDP = 100		MDP = 10	
	Sca.	Par.	Sca.	Par.	Sca.	Par.
CMU-Suite						
Airshed	79.31	71.70	36.55	86.84	96.88	79.86
	(10^2)	(10^2)	(10^1)	(10^3)	(10^9)	(10^3)
fft2	99.72	84.93	99.70	84.90	99.52	84.72
	(10^7)	(10^6)	(10^7)	(10^5)	(10^7)	(10^4)
Radar	71.36	41.12	71.21	42.61	100.00	69.89
	(10^7)	(10^5)	(10^7)	(10^4)	(10^7)	(10^3)
Stereo	84.02	69.81	84.02	69.81	97.68	98.84
	(10^3)	(10^5)	(10^3)	(10^4)	(10^3)	(10^6)
Perfect-Suite						
TRFD	90.40	39.74	53.63	40.78	68.41	47.50
	(10^7)	(10^7)	(10^7)	(10^6)	(10^5)	(10^5)
MDG	74.14	96.56	99.75	92.75	99.60	95.60
	(10^2)	(10^1)	(10^8)	(10^3)	(10^8)	(10^2)
FLO52	61.54	54.62	87.34	51.32	71.05	68.34
	(10^4)	(10^5)	(10^5)	(10^4)	(10^6)	(10^3)
ARC2D	90.21	57.22	55.06	58.37	54.91	59.67
	(10^6)	(10^6)	(10^6)	(10^5)	(10^6)	(10^4)
OCEAN	64.95	57.65	54.44	67.40	53.57	73.05
	(10^6)	(10^4)	(10^7)	(10^3)	(10^7)	(10^2)

Table 1: Scalar and parallel window sizes which accumulated the highest percent of scalar or parallel execution time for MDP = 10, 100 and 1000.

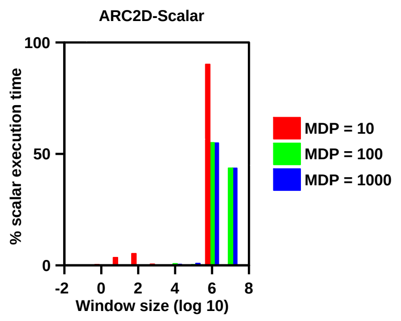

Figure 2: Histogram of the percent scalar execution time with respect to scalar window sizes for ARC2D.

capacity misses. The references that are not covered by the M_hit or M_miss ratios are due to capacity or cold start misses. Table 2 shows a low M_Miss ratio for almost all of the benchmarks. This indicates that the benchmarks also exhibit significant data temporal locality with respect to the degree of parallelism.

3.2 Petaflops Applications

A conveniently programmable machine able to run at speeds of more than 100 TOPS will enable many important applications. Applications which require computation rates of more than 100 TOPS are hereon called "petaflops applications". The significance of this fact is best understood by describing the nature of a subset of applications that fall into two classes: time-constrained, fixed-size problems and large-scale, complex problems.

Time-constrained, fixed-size problems: Programs for time-constrained problems must execute within a given time due to system requirements, are

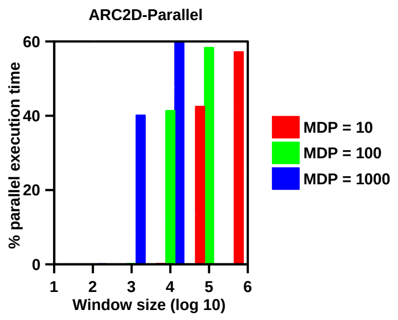

Figure 3: Histogram of the percent parallel execution time respect to parallel window sizes for ARC2D.

compute-bound and have a fixed size dataset. An example of this a class of problems, is scientific simulations used with interactive, immersive visualization systems. These systems allow observers to move freely about computer-generated three-dimensional objects and to explore new environments. Scientist users can then focus on the analysis of the data rather than manipulation of the analysis environment [BiFu 92] by taking advantage of the evolved ability of humans to process three-dimensional spatial information [Ka 93]. In engineering, this technology may be incorporated into the product design cycle, allowing virtual prototyping and testing of products prior to their physical construction. Hence, interactive, immersive three-dimensional visualization is an important medium for scientific applications.

A critical issue facing virtual environments is end-to-end lag time (i.e., the delay between a user action and the display of the result of that action.) Like any closed loop adaptive system, if the lag is too great, users find it difficult to maintain fine control over system behavior and complain that the system is unresponsive. Liu et al. [Li et al. 93] conducted experiments on a telemanipulation system and found the allowable lag time to be 0.1s and 1s for inexperienced and experienced users, respectively.

In [Ta et al.95], a lag model was developed for a finite element simulation executed on an IBM SP [Cr et al. 93]. The following components of the end-to-end lag were measured: tracking, rendering, simulation, data transmission between the simulation system and the display system, and synchronization. The results indicated that the simulation time must be reduced by two orders of magnitude for the visualization system to be considered responsive. Therefore, time-constrained scientific simulations can benefit from a machine capable of achieving petaflops rates. In particular, we consider two applications: finite element and fluid dynamics simulations.

- Finite element simulations are widely used to solve partial differential equations in such areas as circuit simulations, bone deformations and structural mechanics. Implicit finite element methods must solve a system of linear equations, the most computationally-intensive step. Generally, iterative solvers, such as preconditioned conjugate graduate or multigrid, are used since the system matrix is very sparse. Such methods require $N^{1.5}$ floating-point operations, where N is the order of the matrix. For three dimensional problems, grids can range between 10^6 to 10^9 grid points, with six degrees of freedom per grid point. Hence, three dimensional problems can require 10^{14} computations per step. The coupling of the simulations with a visualization system, results in the need to have the 10^{14} computations require less than 0.5s, thereby requiring close to petaflops, performance.

Bench-marks	MDP	Scalar		Parallel	
		M-hit Ratio (%)	M-miss Ratio (%)	M-hit Ratio (%)	M-miss Ratio (%)
CMU-Suite					
Airshed	10	86.4182	13.5340	99.2039	0.7930
	100	99.5312	0.4589	99.8106	0.1856
	1000	99.9867	0.0101	99.8718	0.0932
fft2	10	99.6436	0.2517	97.1807	2.3348
	100	99.6436	0.2517	97.1807	2.3348
	1000	99.6412	0.2541	97.1546	2.3607
Radar	10	99.5987	0.3807	93.3386	6.3088
	100	99.5914	0.3872	92.9609	6.6863
	1000	99.6076	0.3701	88.9526	10.4950
Stereo	10	93.6667	5.5103	93.7090	1.9916
	100	93.6667	5.5103	93.7090	1.9916
	1000	96.3878	2.4318	93.6529	0.9970
Perfect-Suite					
TRFD	10	93.9835	1.2055	99.8709	0.0274
	100	87.6801	93.0352	99.6427	0.0256
	1000	95.0454	3.5792	99.5787	0.3413
MDG	10	84.8500	15.1500	78.9451	21.0539
	100	99.9822	0.0178	99.2532	0.7304
	1000	99.9823	0.0177	99.2471	0.7363
FLO52	10	87.2143	12.6980	99.0314	0.1148
	100	94.9608	3.7852	98.6414	0.5721
	1000	97.7333	1.6851	98.2484	0.7748
ARC2D	10	80.0336	11.1256	92.2167	0.0028
	100	91.8608	2.0135	91.9536	0.0835
	1000	91.5781	1.5383	91.8025	0.1953
OCEAN	10	99.0923	0.9076	97.7900	2.2044
	100	99.3940	0.6060	98.3178	1.6741
	1000	99.3392	0.6600	97.9322	2.0615

Table 2: M-hit and M-miss ratios for $MDP = 10$, 100 and 1000.

- Computational fluid dynamic problems involve computationally-intense simulations used to model the air flow in applications such as weather prediction, combustion modeling and aerodynamics. These applications can greatly benefit from immersive, interactive visualization for virtual prototyping.

Large-scale complex applications: The SPEC High-Performance benchmarks provide a set of large-scale, complex applications [EiHa 96]. One of these applications is based on GAMESS (General Atomic and Molecular Electronic Structure System). Many of the functions found in GAMESS are duplicated in commercial packages used in the pharmaceutical and chemical industries for drug design and bonding analysis.

Seismic is another application from the SPEC benchmarks. It consists of an industrial application representative of modern seismic processing programs used in the search for oil and gas. Seismic is computationally complex and intensive. A typical execution can generate up to a tera-byte of data and requires in excess of 10^{18} floating-point operations. Each execution of Seismic consists of four phases : data generation, data stacking, time migration and depth migration. Phase one is reported not to require any communication. Whereas, both phases two and three require a large amount of communication. Relative to phase two and three, phase four requires a small amount of communication. All the phases include both parallel and sequential sections.

The SPEChpc benchmarks target both current and future high-performance machines. They include scalable data sets. The largest set currently included in the suite exhaust the resources of most available machines. These large data sets do not represent actual limitations of the codes. Even larger sets are available from the code sponsors in SPEC.

Although our initial experiments used small benchmarks, we will use the petaflops applications introduced in this section to further study the different aspects of HPAM.

4 Cost and Performance

Given that petaflops machines will most likely use a large number of components, reducing the cost of these machines becomes a key design issue. In this section we report our preliminary findings about cost-efficiency of HPAM machines and comment on current trends and future technology.

4.1 Preliminary Findings

We made a comparative analysis of a single level multiprocessor with a two-level machine that results from adding a fast processor to the single level. In [AnPo 91] it was shown analytically that the two-level machine can have better cost $\times$ performance than the one level-machine. Our analysis [BeFo 96] with empirical data from the benchmarks introduced in section 3.1 and actual hardware costs indicates the two-level organization is best in almost all the cases.

Table 3 shows the ratios of speedups of a two-level HPAM to a one-level HPAM. The second level of the two-level HPAM and the one-level HPAM have the same number and type of processors. The first level of the two-level HPAM is a fast uniprocessor. In [BeFo 96], it was shown that the two-level HPAM is more cost-efficient than the one-level HPAM if the ratio of their speedups is greater than 1.88, 1.09 and 1.01 for values of MPD = 10, 100 and 1000 respectively. The results of Table 3 show that these conditions are met for almost all the benchmarks and values of MDP. In some cases the gain factor is as high as 8.

The intent of this example was to show that gains can be achieved in terms of speedup and cost-efficiency for 2-level HPAM-like machines using real processors and representative applications. Additional improvement may be achieved by overlapping scalar and parallel computations.

MDP	10	100	1000
CMU-Suite			
Airshed	1.88	7.52	8.14
fft2	7.36	8.06	8.14
Radar	7.81	8.14	8.15
Stereo	2.92	6.62	8.08
Perfect-Suite			
TRFD	1.31	4.15	7.91
MDG	5.36	8.14	8.15
FLO52	1.02	4.65	8.02
ARC2D	1.00	4.65	7.85
OCEAN	6.95	7.13	7.21

Table 3: Ratios of speedups of a two-level HPAM (level 1 : fast uniprocessor, level 2: multiprocessor with MPD processors) to a one-level HPAM multiprocessor with MPD processors identical to the ones in the second level of the two-level HPAM. The results are shown for values of MDP = 10, 100 and 1000.

4.2 Current Trends and Future Technology

There are converging trends in the design of processors and memories that point to future existence of chips that include both processors and memory. Examples include Processors-In-Memory (PIM) [Ko 94], Computational RAM [El et al. 92], IRAM [Pa 95, Sh et.al 96, Sa 96]. Similar ideas were proposed as early as 1970 in [St 70]. The driving argument for these approaches is the fact that the integration of CPU and memory on the same chip brings benefits of lower latency and higher bandwidth in accessing memory that outweigh possible reductions in the complexity of the processor [Sa 96]. Since memory access latency is becoming a limiting performance factor [Jo 95, Wi 95, WuMc 95], it is reasonable to expect that future generations of commercial chips will increasingly follow this trend. In fact, there are already some examples of such chips ([Sa 96, Sh et.al 96, AD 93]).

Two additional trends can be observed in the work mentioned in the last paragraph. One is the inclusion of hardware support for multiprocessor architectures in the integrated CPU-memory chips [Sa 96]. This means that they can be used as the basic blocks for building PAMs. The other is the inclusion of CPU cores on DRAM devices [Sh et.al 96]. It is reasonable to expect that, within 10 years, it will be possible to effectively fabricate chips with several processors and memories (i.e., small PAMs) which can be combined in multichip modules to build (large) PAMs. Chips used to implement different PAM levels can either contain large complex CPUs and small memories or small simple CPUs (or fewer processors) and large memories. Since these chips will be used in general purpose computing their cost should be low enough for their use in large numbers in an HPAM machine. Furthermore, these chips can reuse existing CPU cores that are widely available for commodity parts and thus be less expensive than others that use custom designs.

Currently it is possible to build very efficient shared memory multiprocessor systems with small numbers of processors. Within 10 years, given the above discussion and accepted predictions for semiconductor technology it is reasonable to expect that moderate size multiprocessors with shared memory can be integrated into one or a few chips in a cost-effective manner. However, very large shared memory multiprocessors will continue to present design challenges and will require distributed shared memory implementations. These, in turn, will increase design cost, latencies and processing overheads that will render very large multiprocessors increasingly inefficient unless the requirement of shared memory is relaxed (and implemented in software). A very large one-level shared memory machine capable of more than 100 Teraops would either be too expensive (if we could build it) or would not be able to provide a friendly shared-memory environment with good performance for *all* levels of parallelism. The alternative of choice might be a distributed, message-passing machine. The HPAM approach attempts to provide very efficient cost-effective shared-memory top levels and simpler-to-implement distributed-memory-based lower levels that are also cost-effective but perhaps harder to program. The nature of each level of the HPAM would also change naturally with the evolution of technology. Finally, the HPAM organization lends itself well to single-user mode or multi-users mode at different levels of HPAM. This would amortize costs across several users.

5 HPAM Programming Environment

HPAM can support a programming environment that is usable by users with varying expertise in parallel processing (with proportionate performance returns) by making different levels of the hierarchy user-visible. It can leverage and allow coexistence of evolving practices in parallel programming, including optimizing compilation, data-parallel programming, message-passing and library-based approaches.

A programming environment for the HPAM machine can leverage many tools and software developed for shared-memory and distributed memory machines. There are three possible scenarios for an HPAM programming environment. One scenario relies on automatic translation of a conventional user program into an HPAM program – the user sees only the top level of the HPAM and expects the system to run programs written in standard languages and select PAM levels automatically. The second scenario, a library-based programming approach, would allow users to compose their programs out of existing code blocks which are already optimized for the HPAM machine. The third scenario allows the user to specify how parts of a program should be executed and provide information that the HPAM system could use to allocate code to levels. The HPAM architecture is well-suited for all three scenarios.

The first scenario necessitates an *automatic translation system* which, ideally, is able to analyze program characteristics, the program data set, machine properties, the environment of the program execution and the dynamic machine load. It must be able to assign windows to hierarchy levels statically at compile time, at runtime and also facilitate dynamic migration.

For window mappings, several tradeoffs have to be considered such as parallelism versus execution speed of individual processors, inter-level versus intra-level communication costs and the cost of migrating windows versus the load-imbalance of fixed window assignments.

The decision about window mappings can be made at compile time if sufficient information can be obtained from the source program. Decisions will be deferred to runtime where additional information on the actual data set and the current load of the machine is available.

Our implementation of the translation system will be based on the Polaris compilation system. An initial static algorithm will use Polaris' symbolic analysis capabilities to gather information from the program that leads to static mapping decisions. In a second step the compiler will be able to inject code into the program that evaluates the best mapping at runtime. Furthermore, Polaris includes an infrastructure for adaptive and speculative program analysis methods, which will be available for window definitions and mappings. These methods can use knowledge from the program execution history and execute program segments speculatively with a possible backtracking step if the speculation failed.

The second programming scenario will facilitate the composition of programs from *optimized libraries*. To this end a study and implementation of the algorithms used in the application programs in the form of library routines is needed. This involves identifying program segments that can be optimized as individual subroutines and characterizing their behavior on the HPAM architecture.

For the *knowledgeable programmer* automatic translation tools are a starting point. In addition performance instrumentation, analysis and visualization tools are needed. These tools will allow the programmer to observe the detailed program behavior and tune it to a given HPAM machine.

6 Conclusions and Future Work

This study presented an initial high-level description and motivation for a 100-Teraops multilevel heterogeneous machine (HPAM). However, more detailed studies are needed *to refine and analyze* this design by carefully considering its technological feasibility along with the usability of the programming environment in the context of important representative applications.

Future work includes understanding technology advances and how they affect an HPAM architecture. The challenge is to compare a traditional multiprocessor design which is based on the technology available in 10 years to an HPAM that consists of components that might span several generations of processors and memories. Furthermore, understanding how different parameters of HPAM (such as number of processors, size of memory, and memory latency per level) affect the execution of petaflop applications is a crucial component of our design strategy.

References

[AD 93] ADSP-21060 SHARC *Super Harvard Architecture Computer*, ANALOG DEVICES, Norwood, MA, Oct. 1993.

[AnPo 91] J.B. Andrews and C.D. Polychronopoulos, "An Analytical Approach to Performance/Cost Modeling of Parallel Computers," *J. Par. Dist. Computing,* v. 12, n. 4, Aug. 1991, pp. 343-356.

[Be et al. 94] M. Berry, D. Chen, et al., "The Perfect Club Benchmarks: Effective Performance Evaluation on Supercomputers," CSRD, Univ. of Illinois, Urbana-Champaign, Illinois, Tech. Rep. UIUC-CSRD-827, Jul. 1994.

[BeFo 96] Z. Ben-Miled and J.A.B. Fortes, "A Heterogeneous Hierarchical Solution to Cost-efficient High Performance Computing," *seventh IEEE symp. on Par. and Dist. Processing*, 1996.

[BiFu 92] G. Bishop and H. Fuchs, "Research Directions in Virtual Environments," *ACM Computer Graphics,* v. 26, n. 3, 1992, pp. 153-177.

[Bl et al. 94] W. Blume, R. Eigenmann, et. al., "Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing," *IEEE Parallel and Distributed Technology,* v. 2, n. 3, Fall 1994, pp. 37-47.

[Cr et al. 93] C. Cruz-Neira, D.J. Sandin, and T.A. DeFanti, "Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE," *Proc. SIGGRAPH,* v. 27, Aug. 1993, pp. 135-142.

[Di et al. 94] P.A. Dinda, T. Gross, et. al., "The CMU Task Parallel Program Suite," School of Computer Science, Carnegie Mellon Univ., Tech. Rep. CMU-CS-94-131, Jan. 1994.

[EiHa 96] R. Eigenmann and S. Hassanzadeh, "Benchmarking with Real Industrial Applications: The SPEC High-Performance Group," *IEEE Computational Science and Engineering*, v. 3, n. 1, Spring 1996, pp. 18-23.

[El et al. 92] D. Elliot, W.M. Snelgrove et. al., "Computational RAM: A Memory-SIMD Hybrid and its Application to DSP," *Proc. Custom Integrated Circuits Conf,* May 1992, pp. 30.6.1-30.6.4.

[Jo 95] E.E. Johnson "Graffiti on "The Memory Wall"," *ACM Computer Architecture News,* v. 23, n. 4, Sep. 1995, pp. 7-8.

[Ka 93] R.S. Kalawsky, "The Science of Virtual Reality and Virtual Environments," Addison-Wesley Publishers, 1993.

[Ko 94] P.M. Kogge, "EXECUBE - A new architecture for scalable MPPS," *1994 International Conference on Parallel Processing,* Aug. 1994, pp. I.77-I.78.

[Li et al. 93] A. Liu, G. Tharp, et. al., "Some of What One Needs to Know about Using Head-Mounted Displays to Improve Teleoperator Performance," *IEEE Trans. Robotics and Automation,* Vol. 9, 1993, pp. 638-648.

[MeAl 90] D. Menasce' and V. Almeida, "Cost-performance Analysis of Heterogeneity in Supercomputer Architectures," *Proc. Supercomputing Conf.,* Nov. 1990, pp. 169-177.

[Mo et al. 96] D. Moncrieff, R. E. Overill, et. al., "Heterogeneous Computing Machines and Amdahl's Law," Parallel Computing, v. 22, n. 3, 1996, pp. 407-413.

[Pa 95] D.A. Patterson, "Microprocessors in 2020," *Scientific American,* v. 273, Sep. 1995, pp. 62-65.

[Sa 96] A. Saulsbury, F. Pong, et. al., "Missing the Memory Wall: The Case for Processor/Memory Integration," *Proc. 23rd Int. Computer Architecture Symp.,* Jun. 1996, pp. 90-101.

[Sh et.al 96] T. Shimizu, J. Korematu, et. al., "A Multimedia 32b RISC Microprocessor with 16Mb DRAM," *Proc. Int. Solid-State Circuits Conf.,* Feb. 1996, pp.216-217.

[St 70] H.J. Stone, "A Logic-in-Memory Computer," *IEEE Trans. in Computers,* v. 14, n. 3, Jan. 1970, pp. 73-78.

[St et. al 95] T. Sterling, P. Messina, P. H. Smith, "Enabling Technologies for Petaflops Computing," *MIT press*, 1995.

[Ta et al.95] V.E. Taylor, M. Huang, et. al., "Performance Modeling of Interactive, Immersive Virtual Environments for Finite Element Simulations," to appear *Int. J. on Supercomputing Applications*, Issue 10.2, Spring 1996.

[Wi 95] M.V. Wilkes, "The Memory Wall and the CMOS End-Point," *ACM Computer Architecture News,* v. 23, n. 4, Sep. 1995, pp. 4-6.

[WuMc 95] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *ACM Computer Architecture News,* v. 23, n. 1, Mar. 1995, pp. 20-24.