Programming Models and System Software for Future High-End Computing Systems: Work-in-Progress

Guang R. Gao, Kevin B. Theobald, R. Govindarajan, Clement Leung, Ziang Hu,
Haiping Wu, Jizhu Lu, Juan del Cuvillo, Adeline Jacquet, Vincent Janot
Department of Electrical and Computer Engineering
University of Delaware, Newark, DE, 19711
{ggao, theobald, govind, hu, hwu, jlu, jcuvillo, janot}@capsl.udel.edu

Thomas L. Sterling
Center for Advanced Computing Research
California Institute of Technology, Pasadena, CA 91125A
tron@cacr.caltech.edu

Abstract

Future high-end computers which promise very high performance require sophisticated program execution models and languages in order to deal with very high latencies across the memory hierarchy and to exploit massive parallelism. This paper presents our progress in an ongoing research toward this goal. Specifically, we will develop a suitable program execution model, *a high-level programming notation which shields the application developer from the complexities of the architecture, and a compiler and runtime system based on the underlying models. In particular, we propose fine-grain multithreading and thread percolation as key components of our program execution model. We investigate implementing these models and systems on novel architectures such as the HTMT architecture and IBM's Blue Gene. Also, we report early performance prediction of* thread percolation *and its impact on execution time.*

1 Introduction

Future high-performance computers, such as IBM Blue Gene, the ASCI teraflops-class systems, and the architecture(s) proposed in the NSF petaflops point-design studies, will have thousands to millions of processors, and complex multilevel memory hierarchies with memories physically distributed across the machine. To use such machines effectively, enormous amounts of parallelism must be exposed in programs, and careful attention must be paid to the latency and bandwidth of access to different levels of memory. Applications enabled by such high-end machines are also expected to be significantly more complex, irregular and dynamic than today, presenting a major software challenge. This position paper discusses novel software technologies which will be needed to meet these challenges.

The challenges in programming high-end next generation computer architectures can be met by novel software technologies for the design, development, runtime support, and management of applications. Major components in accomplishing this goal are (i) design and implementation of a suitable program execution model, (ii) design and development of a high-level programming notation which shields the application developer from the complexities of the architecture wherever possible, and (iii) implementation of a compiler and runtime system which enable the efficient execution of such high-level programs under this program execution model. The program execution model should be the foundation for an integrated software environment that eliminates unnecessary boundaries between the different components and layers, and enhances the manageability of the entire system. The programming notation should not require radically different programming paradigms in different parts of a given machine; moreover, it should not demand that users significantly rewrite their applications for each new machine. Finally, the compiler and runtime system technology should be powerful enough to deduce information about program behavior which is required for efficient execution but which is not specified explicitly in a high-level application program.

In this paper, we discuss our work toward a novel program execution model and its corresponding programming notation. The program execution model is a refinement of the current EARTH multithreaded system [10, 19], mod-

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)

ified to support the types of supercomputer architectures envisioned for this study and embellished with *thread percolation*, an aggressive prefetch-like technique to reorganize code/data to a proper space/level in the memory hierarchy before a program unit (such as a procedure instance) starts execution. The refined model features fine-grain multithreading, support for a shared address space with atomic synchronizing memory operations, and dynamic load balancing.

In this paper, we describe a novel executable analytical performance prediction approach for high-end architectures which is helpful in evaluating certain features of the underlying architecture/program execution model quickly, even before their designs become concrete and/or support software (such as runtime system or compilers) become available. Using the performance predication approach we evaluate thread percolation in an architecture with deep memory hierarchy. This paper concludes by discussing the compiler and runtime system support for these models, with a specific emphasis on the system software support for IBM Blue Gene machine.

2 Background and Challenges

In this section, we give a high-level overview of the architectures of future high-end platforms and enumerate challenges for next-generation software.

2.1. High-end Architectures

Future generations of high-end architectures will have thousands to millions of processors interconnected with multiple levels of networks, and a complex multilevel memory hierarchy with physically dispersed memories. At least three classes of architectures have been studied by us in some depth:

Revolutionary architectures exploit radically new hardware technologies outside the mainstream of commercial development. One example is the HTMT (Hybrid Technology Multi-Threaded) Architecture [7], in which a (relatively) modest number of 100GHz superconducting processors are employed with a memory hierarchy of at least 4 levels. HTMT memory latencies increase by orders of magnitude from one level to the next, and these latencies are tolerated without loss of efficiency by multithreaded processor architectures.

Evolutionary architectures track current trends in mainstream computer evolution, both in off-the-shelf processors and networking technology, leveraging the huge investments in commercial computing systems. Examples include the ASCI teraflops-scale machines [15] and Chiba City cluster architectures [1], expected to reach several hundred teraflops. Such systems will have several hundred thousand to several million processors.

Novel architectures include machines being developed under the IBM Blue Gene and NASA Gilgamesh projects. They use conventional hardware technologies, but use custom processors rather than relying on off-the-shelf components. In our work we specifically study the IBM Blue Gene architecture [3], which is based on a Cellular architecture, and possible implementation of our advanced program execution models on this architecture.

2.2. Challenges to System Software

We believe the most important challenge for high-end parallel machines is the management of latency and bandwidth, made difficult by the deep memory hierarchy and the complex applications that need dynamic management. We address this problem with a three-pronged approach:

1. First, a *program execution model* (PXM) is specified, as a solid foundation for the proposed software technologies for high-end architectures with complex latency tolerance and management.

2. Second, a low-level *programming model* is developed based on the proposed PXM. This model will explicitly expose the memory hierarchy and the performance cost of data or computation movements within the hierarchy. Moreover, it will provide programmable abstractions that can facilitate the latency/bandwidth management to achieve desirable performance.

3. Finally, high-level languages, compilers and run-time systems are required, as discussed in later sections. Together, these components will provide the basis for developing and managing complex applications.

3 Program Execution Model

The program execution model (PXM) is the basic low-level abstraction of the underlying system architecture upon which our programming model, compilation strategy, runtime system, and other software components are developed. The PXM serves as an interface between the architecture and the software. Unlike an instruction set architecture (ISA) specification, which usually focuses on machine-specific low-level details such as opcodes and registers, the PXM refers to components at a higher level for the whole family of high-end machines (complemented by an ISA specification of each particular machine). The key features necessary to support the PXM are assumed to be implemented directly in hardware or indirectly by a combination of available hardware features and runtime system support. The key components in our program execution model are

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)

fine-grain multithreading and thread percolation which we discuss in detail below. Also we briefly describe the memory model, synchronization, and dynamic load balancing features of the PXM in this section.

Thread model: Like other multithreaded execution models, our model divides a program into multiple sections, or threads. However, a distinct feature of our thread model is the mechanism used to form, enable, schedule, and synchronize threads. Our thread model comprises two layers. The first, *threaded function invocation,* forks a new thread to execute another function while the caller continues execution. This is an important source of parallelism. When a threaded function is invoked, a *frame* of storage is allocated from memory for its local context. Our own experience (with EARTH [9, 18, 20, 22]) and that of others (e.g., Cilk [4]) indicates that lightweight threaded function invocation enhances both the programmability and the efficiency of parallel applications.

The second layer of threads in our base PXM involves *fibers* — ultra-lightweight threads within the body of a threaded function. A fiber can run as soon as its data and control dependences are satisfied. Compiler and runtime system support will ensure the low overhead of fiber initiation and termination. Fibers provide several benefits, including providing a basis for implementing split-phase memory operations and exploiting fine-grain parallelism.

Thread percolation: A key feature of our proposed model is the explicit exposition of the complex memory hierarchy and of the computation and data movement costs to the programming model of high-end machines. This transparency will be achieved by providing programmable abstractions that enable efficient management of the latency/bandwidth across the system. To this end, we develop a new strategy, called *thread percolation*. Percolation is an aggressive prefetch-like technique. However, unlike prefetching, which involves moving blocks of data within the memory hierarchy, thread percolation manages contexts, which include data, program instructions, and control state. Moreover, whereas in most prefetching mechanisms a prefetch starts only when a processor issues a request, in thread percolation the main processors might be unaware of the preparation of many contexts throughout the system.

Under the percolation model, the application can steer both the program flow and the data movement across the memory hierarchy to cause the data to "meet" the corresponding threads *just in time* at the vicinity of the processors where the computation is to be carried out. In other words, the percolation dictates an "active caching" architecture in order to optimally match the behavior of the memory system to the application needs, and to rapidly adapt to changes in the application's data locality characteristics. The enabling of a thread may involve either gathering the data toward the processor(s) where the thread is enabled, called *inward percolation*, or sending/migrating the thread toward the vicinity of the data, called *outward percolation*. This percolation model builds on our experience with an earlier percolation model developed for the Hybrid Technology Multi-Threaded (HTMT) architecture [6, 8]. The percolation model extends dynamic prefetching to allow the management of contexts that include data, program instructions, and control states. We have evaluated the performance improvement due to percolation through an analytical performance evaluation study which is reported in Section 7.

Memory model: Our PXM will support *global addressing* through a combination of architecture features and compiler/runtime support. Global addressing makes naming logically shared data much easier for the programmer because any thread can directly reference any data in the shared space and the naming model is similar to that on a uniprocessor. Additionally, global naming can improve the programmability and efficiency of applications with irregular and unpredictable data needs [17, 23] and can facilitate support of dynamic load balancing of threads. One refinement in our model is a frame-based addressing scheme, where global addresses are formed as a combination of global frame pointers and local offsets. This scheme yields many benefits, such as simplifying memory analysis for the compiler and permitting more ambitious dynamic load-balancing.

Synchronization and scheduling: Under our PXM, the data and control dependencies among fibers (within the same function) and among threaded function invocations (via their respective fibers) will be made explicit. A synchronization signal will be posted from one fiber to another, either in the same or in another threaded function instance, to inform the recipient that a specific dependence has been satisfied. Our PXM provides atomic operations for sending data (possibly to a global memory location) and posting an event, to guarantee that the data has been properly transferred before fibers that use that data are run.

Dynamic load balancing: Under the base PXM, a threaded function can be declared "movable" by the programmer or compiler, allowing the runtime system to run that function on any node. This permits the use of dynamic, low-overhead load-balancing algorithms [2, 10, 12].

4 Programming Model and Threaded-C

Once the proposed program execution model has been developed and specified, the next step is to develop a low-level programming model that can serve as a foundation for

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)

system software. The simplest approach is to present the PXM operations directly in the language. A program in this language might be produced by a compiler or directly by a programmer. We propose to extend EARTH Threaded-C [19, 21] (a standard ANSI-C with threaded operations) to accommodate our proposed PXM.

The initial specification of our new Threaded-C will begin with modification of the following features from EARTH Threaded-C: the threaded function specification, the fiber specification (represented as threads within a function body in EARTH Threaded-C), the *sync_slot* mechanism that enforces fiber and thread synchronization, the set of split-phase operations such as *block_move_sync* and *data_sync*, and the *token* and *invoke* mechanism to control static and dynamic load balancing [19, 21]. We will then add new mechanisms to (1) start the preparation of data for threads that will be enabled in the near future; (2) move data and code among memory regions; (3) pack data and code together to enable their joint percolation; and (4) enable the management of memory space to store data waiting to be percolated, processed, or disposed.

Percolation presents two challenges for development of the programming model. The first is to provide user-friendly constructs in the language to express the physical locality requirements of a threaded function. For this we introduce *parcels*. A parcel encapsulates both the threaded code and the data required to ensure the desired layout for execution.

The second challenge is to organize the complex memory hierarchy so that the data movement and reorganization dictated by thread percolation can be programmed effectively. High-end architectures, such as those proposed for HTMT and Blue Gene, do not have conventional data caches. Rather, each level of memory is considered as a "buffer" of the next level; these buffers are directly addressable. Buffer storage allocation and data movement at each level can be fully or partially under the programmer's control. Moreover, since the penalty of a memory access miss that is in the critical path of a program execution may be severe, we will need to develop schemes that do not rely on a "hidden" automatic coherence management mechanism.

We propose to extend the EARTH Threaded-C language with a set of primitives to implement percolation. Potential candidates include primitives to copy and move blocks of data and dispatch parcels from one level to another in the memory hierarchy and to allocate and release space for a block within the current memory level. Further, the language will require additional programming mechanisms to support data reorganization (e.g., gather/scatter, pointer swizzling, generalized corner turn), some of which might be best implemented as library functions.

Finally, we need to extend Threaded-C and its programming model so the users can express their computation to take advantage of both thread percolation and dynamic load balancing *together*. The concrete programming notations required, and their semantics and implementation issues, are a part of the proposed research.

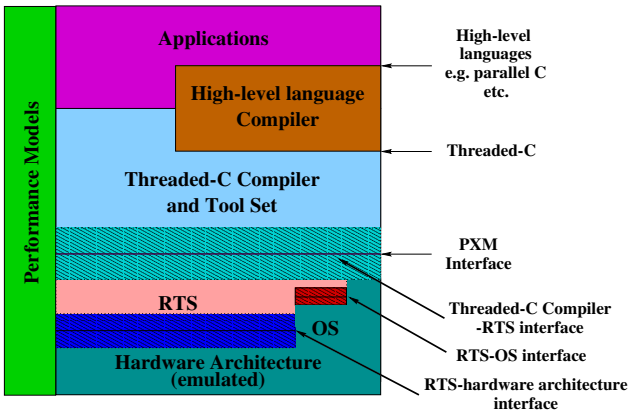

Figure 1. Software architecture

5 System Software Support

The proposed programming and program execution model will provide the foundation for our system software. Included are a number of closely integrated software components: compiler, runtime system (RTS), resource and application management tools, performance debugging, profiling, evaluation tools guided by suitable performance models, and a simulation/emulation platform. Together they provide a basis for developing and managing complex applications.

The proposed software framework is illustrated in Figure 1. One unusual feature is that there is no fixed boundary between different software layers: the shaded regions represent close interactions between the Threaded-C compiler and RTS, and between the RTS and the architecture layer. Moreover, the performance models and tools are integrated across all layers. Our research will focus on the runtime percolation model and memory management.

5.1. Runtime system

The essential features of the RTS are to implement PXM which will interact with the architecture layer and the Threaded-C compiler. The RTS will also provide resource and application management with the OS layer. The implementation of the RTS for PXM leverages our experience with the design of the EARTH runtime system. However, it needs to be extended to meet deep memory hierarchies found in future high-end systems which are likely to be shared, and may be explicitly divided into multiple levels, leading to non-uniform address model at each level.

We study two options for addressing memory across remote nodes: using a *global address* directly to name a location, or using a *frame-based address*, since most data are encapsulated with frames associated with corresponding threaded function invocation.

We envision that runtime system for percolation will comprise three major components to support percolation: the parcel invocation and termination module (PIT), the parcel assembly and disassembly module (PAD), and the parcel dispatcher and dispenser module (PDD). The PIT invocation manager will invoke enabled parcel functions and send them to the PAD for processing. The PAD assembly manager will move the required code and data into local memory locations. The parcel will then be passed to the PDD, which will reserve space in the fast (local) memory region and then move the data and code associated with the parcel into the reserved region. After parcel execution, the PDD dispenser manager will copy the buffered data to the slower memory level and deallocate the resource reserved by the parcel. Then the PAD disassembly manager will disassemble and distribute output data into its proper places. When the disassembly process is finished, the PIT termination manager will inform the dependent successor parcels that the parcel under consideration has completed execution. This may cause its successors to become enabled, thus beginning another percolation process. Challenging issues remain as how to optimize percolation, how to parallelize the process, and how to manage the tradeoff between latency hiding and bandwidth requirement with different memory levels.

5.2. Compilers

The compiler must work closely with the runtime system and must heavily interact with the system resource management and the performance models of the related software and hardware components.

Threaded-C compilation: We have identified four key issues regarding Threaded-C compilation:

1. Code partitioning into fibers for optimal performance. The compiler must have access to cost models of the underlying architecture and RTS.

2. Register allocation and synchronization resources. This issue, which is critical to code efficiency, must be investigated in conjunction with development of the dynamic RTS fiber scheduling policy.

3. Compilation for percolation. Techniques are needed for analyzing the data and code movement required by a parcel, minimizing unnecessary copying and maximizing reuse. Optimization requires close interaction between compiler, RTS, and the performance models at both the architecture and the software levels.

4. Automatic relocation and automatic caching mechanisms. Efficient algorithms must be devised for these automatic mechanisms.

Our proposed compilation strategy is based on the *data-centric compilation technology* developed by Pingali and co-workers [13]. Conventional restructuring compilation technology is *control-centric*: it reasons directly about the control flow of the program and reorders this control flow to accomplish goals like parallelization or locality enhancement. This technology works well for perfectly nested loops, but extending it beyond perfectly nested loops appears problematic. Even for relatively simple programs like Cholesky factorization, code generated by conventional compilers performs 5 to 10 times worse than hand-coded libraries such as LAPACK [14]. Data-centric code generation addresses some of these problems. Rather than manipulating the control structure of the program directly, the compiler determines the order in which data elements should be fetched into the highest level of the memory hierarchy, and then schedules close together statement instances that touch a given data item. In principle, this scheduling can be done across imperfectly nested loops and even across procedure boundaries. Experiments at SGI indicate that, for dense numerical linear algebra applications, data-centric technology improves performance by factors of 2 to 5.

6 Case Study: Cellular Multiprocessor-on-a-Chip Architecture

In this section, we present a work-in-progress report on a case study using a class of Cellular multiprocessor-on-a-chip architecture. An example of such a Cellular architecture has been reported in [3]. Our discussion below, however, will use a generic Cellular architecture model to allow the generality and flexibility in the development of program execution models and system software technology for this class of architectures. In the following subsection we describe the architecture. Sections 6.2 and 6.3 deal with PXM and system software support for Cellular architecture.

6.1. Architecture

The IBM Blue Gene is based on Cellular architecture [3],. Each chip contains a number of processors and each processors consists of multiple thread units, and a single floating point unit. Each processor is a simple (in-order issue) processor operating at a moderate clock rate. The memory hierarchy of the Blue Gene architecture is deep, having registers, caches, on-chip and off-chip memories.

More specifically, within a chip, there are multiple memory banks and the processors in a chip have access to them. The memory banks together form an on-chip shared memory for the processors. Lastly, there is an off-chip memory associated with each chip. The off-chip memory of all chips form a distributed memory/address space. The program execution model supported is shared memory programming within a chip and message passing across chips.

6.2. Program Execution Model

In this section we discuss a few possible implementation of our program execution model on the Cellular architecture. The first implementation of our PXM is based on a pthread-like model with direct mapping of software threads into thread units. The initial implementation will consider mapping a single thread to a hardware thread unit. Upon initialization, a software thread is given control over a well determined region of the on-chip memory, which is allocated to every physical thread unit at boot time. This enables fast thread creation and reuse. As in the pthread model, a waiting thread (waiting on an external event/synchronization) goes to sleep; such a thread is woken up by another thread through a hardware interrupt/signal.

We are also considering a processor-centric approach where a small number of software threads are mapped into a processor consisting of multiple thread units. Thread units within a processor are now available to run user code, which can be seen either as a single thread or as a set of data-dependency-related fibers, and provide support for specific tasks such as inter-thread/fiber synchronization and communication or percolation. This will facilitate implementing the fine-grain multithreaded model discussed in Section 3. The proposed model will also enable us to implement dynamic thread spawning and load balancing on the hardware thread units.

6.3. System Software Support

We use the Open64 compiler infrastructure to study compilation techniques suitable for Cellular supercomputers. This compiler infrastructure will be retargetted to the instruction set of the Cellular architecture. It will also support a run-time library designed according to our program execution model. We will explore the program representations and manipulation algorithms to develop optimization techniques tailored to the Cellular architecture.

To allow applications software development and performance evaluation, we are also building a software simulator for the Cellular architecture, and a complete software development tool suite including a loader, a linker and a debugger. We are also developing a low-cost FGPA-based emulator to execute and evaluate larger applications programs. This work on system software is currently under progress.

7 Early Performance Trend Prediction

In this section, we describe our novel performance prediction approach and its applications on evaluating the impacts of percolation.

7.1. Executable Analytical Performance Model

In this section, first we describe our novel *executable* analytical performance evaluation approach which is useful in predicting performance trends in early stage architecture design space exploration. The executable analytical performance evaluation model can evaluate architectures over a wide operating range and is helpful in identifying certain architectural features that are advantageous even before their designs become concrete and system software support (e.g., runtime system or compiler) available. Analytical performance evaluation approach also has the advantage that they are relatively less time consuming than the simulation approach.

Another novel feature of this work is that it models the architecture (hardware) and the software (program execution) components and the interaction between them. To accomplish this, we model program execution through a program graph that models coarse-grain parallelism in the application. The program graph is *executed* on the architecture model. The resulting analytical model is solved using a queuing network tool enriched with synchronization [16]. The executable queuing network model executes the (abstract) program model on the architecture to obtain performance metrics. This approach, referred to as the executable approach, simulates the execution of the program model on the architecture model. The proposed approach is very effective in obtaining performance predictions for large complex systems and for a wide operating range fairly quickly.

7.2. Architecture and Program Model

To evaluate the impact of percolation, we have considered a an architecture with a single processor, with different levels of memory hierarchy, and support for percolation as shown in Figure 2. In order to support multithreading, we assume that there is a Synchronization Processor (SP), whose role is to ensure that only computation units which are data ready are enabled for execution. In our initial model, we assume that each invocation of a procedure (referred to as a *procedure instance*) is a computation unit. (Each procedure instance consists of a number of threads. In our initial program model, we have assumed that each procedure instance p has $Size(p)$ sequential threads, where $Size(p)$ follows certain distribution with a mean 6 [11].)

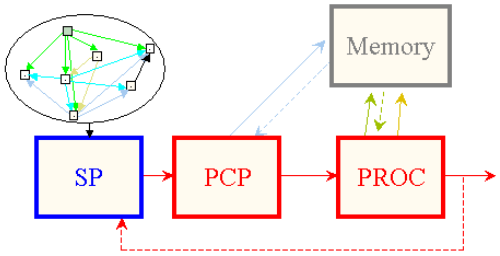

Figure 2. Architecture Considered

Hence enabled procedure instances are sent to a Percolation Co-Processor (PCP). The role of the PCP is to send requests to the memory to percolate data that will be accessed by the procedure instance to higher levels in the memory hierarchy. Not all data can be percolated as for some data their addresses may be computed at runtime during the execution. When all the data requests for a given procedure instance are percolated, the procedure is said to be $ready$ for execution, and moves to the processor.

To model percolation, we assume that certain amount of data ($pcache$) that is required for the execution of a procedure instance is already available in a high-speed software managed cache. The remaining data ($1 - pcache$) must be accessed from the memory. For the data not available in the cache, percolation is employed. As mentioned earlier not all data can be percolated. We denote $pperco$ as the part of data that can be percolated. For the remaining data that cannot be percolated, during the execution of a procedure instance, data requests are sent to memory by the processor (PROC). We assume that the processor is multithreaded and can switch context to another thread instead of waiting for a memory request to be satisfied. The thread which has sent a memory request, waits in a Wait Queue, until the memory request is satisfied. After that, the thread joins the Ready queue and competes for processor resource. The memory has $nbports$ each to serve percolation requests as well normal data requests. The analytical model for the architecture is shown in Figure 3.

We model a program as a set of procedure instances, characterized by a name and a list of dependencies. We refer to this as a program graph which consists of nodes, representing dynamic instances of procedures, and directed arcs connecting the nodes indicating dependencies between procedures. The program graph is acyclic. As mentioned earlier, each procedure instance consists of a number of sequential threads. We use randomly generated program graphs as inputs to our model. Using a queuing network simulation tool, we simulate the queuing models. In our simulation we assume that the latencies of the cache and the memory as 2 and 100 cycles respectively. The value of $pcache$ and $pperco$ are kept as parameters in our experiments.

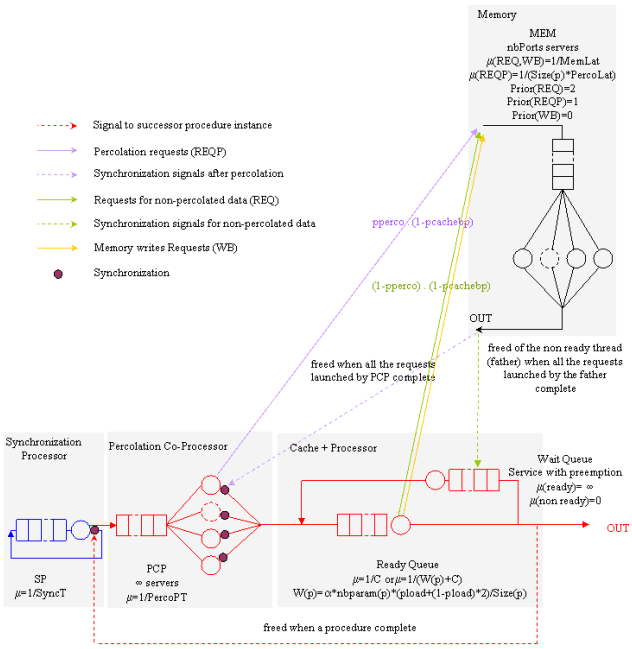

Figure 3. Analytical Model

7.3. Performance Results

We observe that percolation improves the execution time in a multithreaded architecture. The multithreaded architecture which allowed us to hide a large part of the latency, percolation brings an additional improvement upto 20% (refer to Figure 4). Lower the $pcachebp$ value, higher is the

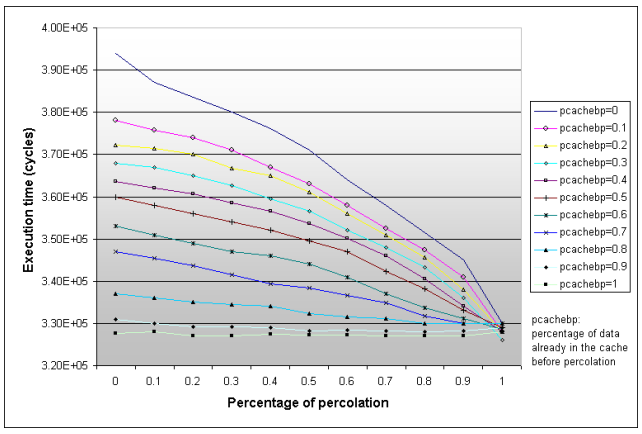

Figure 4. Performance results (Execution Time) for percolation in a Multithreaded Architecture

improvement in performance. That is, percolation brings increasingly significant performance gains when the cache hit ratios (before percolation) are low We observe that percolation and multithreading nicely complements each other

to improve the performance. We have also conducted studies on the impact of percolation on multithreading multiprocessor architectures. Details of these models and their performance results can be found in [11].

8 Other Work

In addition to the progress reported in this paper, the Cornell group and the University of Chicago group have made further progress in their work. These will be reported subsequently in the annual report.

References

[1] Argonne National Laboratory. Chiba City, the Argonne Scalable Cluster URL http://www-unix.mcs.anl.gov/chiba/.

[2] H. Cai. Dynamic load balancing on the EARTH-SP system. Master's thesis, McGill U., Montréal, May 1997.

[3] C. Cascaval, J. G. Castaños, L Ceze, M. Denneau, M. Gupta, D. Lieber, J. E. Moreira, K. Strauss, and H.S. Warren, Jr. Evaluation of a multithreaded architecture for Cellular computing. In *Proc. of 8th Intl. Symp. on High-Performance Computer Architecture*, pages 311–321, Boston, Mass., Feb. 2002.

[4] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. of SIGPLAN PLDI '98*, pages 212–223, Montréal, Jun. 1998.

[5] *Proc. of Frontiers '96: The Sixth Symp. on the Frontiers of Massively Parallel Computation*, Annapolis, Mary., Oct. 1996.

[6] G. Gao, J. N. Amaral, A. Marquez, and K. Theobald. A refinement of the HTMT program execution model. CAPSL Tech. Memo 22, Dept. of Elec. and Computer Eng., U. of Delaware, Newark, Del., Jul. 1998. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[7] G. Gao, K. K. Likharev, P. C. Messina, and T. L. Sterling. Hybrid technology multi-threaded architecture. In *Proc. of Frontiers '96* [5], pages 98–105.

[8] G. R. Gao, K. B. Theobald, A. Márquez, and T. Sterling. The HTMT program execution model. CAPSL Tech. Memo 09, Dept. of Elec. and Computer Eng., U. of Delaware, Newark, Del., Jul. 1997. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[9] G. Heber, R. Biswas, P. Thulasiraman, and G. R. Gao. Using multithreading for the automatic load balancing of adaptive finite element meshes. In *Proc. of the 5th Intl. Symp. on Solving Irregularly Structured Problems in Parallel*, number 1457 in LNCS, pages 132–143, Berkeley, Calif., Aug. 1998. Springer-Verlag.

[10] H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, G. R. Gao, and L. J. Hendren. A study of the EARTH-MANNA multithreaded system. *Intl. J. of Parallel Programming*, 24(4):319–347, Aug. 1996.

[11] A. Jacquet, V. Janot, C. Leung, G.R. Gao, R. Govindarajan, and T.L. Sterling. An exeuctable analytical performance evaluation approach for early performance prediction. In *Proc. of the Workshop on Masssively Parallel Processing*, Nice, France, Apr. 2003.

[12] K. P. Kakulavarapu. Dynamic load balancing issues in the EARTH runtime system. Master's thesis, McGill U., Montréal, Apr. 2000.

[13] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multilevel blocking. In *Proc. of SIGPLAN PLDI '97*, pages 346–357, Las Vegas, Nev., Jun. 1997.

[14] I. Kodukula, K. Pingali, R. Cox, and D. Maydan. An experimental evaluation of tiling and shackling for memory hierarchy management. In *Conf. Proc., 1999 Intl. Conf. on Supercomputing*, pages 482–491, Rhodes, Greece, Jun. 1999.

[15] Lawrence Livermore Laboratory. Accelerated strategic computing initiative (asci). URL http://www.llnl.gov/asci/.

[16] Simulog France. *QNAP - Queuing Tool*. http://www.simulog.fr/.

[17] J. P. Singh, J. L. Hennessy, and A. Gupta. Scaling parallel programs for multiprocessors: Methodology and examples. *Computer*, 26(7):42–50, Jul. 1993.

[18] A. Sodan, G. R. Gao, O. Maquelin, J.-U. Schultz, and X.-M. Tian. Experiences with non-numeric applications on multithreaded architectures. In *Proc. of the 6th PPOPP*, pages 124–135, Las Vegas, Nev., Jun. 1997.

[19] K. B. Theobald. *EARTH: An Efficient Architecture for Running Threads*. PhD thesis, McGill U., Montréal, May 1999.

[20] K. B. Theobald, G. Agrawal, R. Kumar, G. Heber, G. R. Gao, P. Stodghill, and K. Pingali. Landing CG on EARTH: A case study of fine-grained multithreading on an evolutionary path. In *Proc. of SC2000*, Dallas, Tex., Nov. 2000.

[21] K. B. Theobald, J. N. Amaral, G. Heber, O. Maquelin, X. Tang, and G. R. Gao. Overview of the Threaded-C language. CAPSL Tech. Memo 19, Dept. of Elec. and Computer Eng., U. of Delaware, Newark, Del., Mar. 1998. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[22] K. B. Theobald, R. Kumar, G. Agrawal, G. Heber, R. K. Thulasiram, and G. R. Gao. Developing a communication intensive application on the EARTH multithreaded architecture. In *Proc. of Euro-Par 2000*, number 1900 in LNCS, pages 625–637, Munich, Germany, Aug.–Sep. 2000. Springer-Verlag.

[23] M. S. Warren and J. K. Salmon. A parallel hashed oct-tree N-body algorithm. In *Proc. of Supercomputing '93*, pages 12–21, Portland, Ore., Nov. 1993.