A Heterogeneous Lightweight Multithreaded Architecture

Sheng Li[1], Amit Kashyap[1], Shannon Kuntz[1], Jay Brockman[1], Peter Kogge[1], Paul Springer[2], and Gary Block[2]

[1]University of Notre Dame
Department of Computer Science and Engineering
Notre Dame, IN 46556
{sli2, akashyap, skuntz, jbb, kogge}@nd.edu

[2]NASA Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109
{Gary.L.Block, pls}@jpl.nasa.gov

Abstract

Programs with irregular patterns of dynamic data structures and/or those with complicated control structures such as recursion are notoriously difficult to parallelize efficiently. For some highly-irregular applications, such as a SAT solver, it has been nearly impossible to obtain significant parallel speedups on conventional SMP systems over serial implementations. Lightweight multithreading, as found in the Cray MTA and the upcoming XMT (Eldorado), has been demonstrated as an effective approach to attacking these problems. In this paper, we describe a heterogeneous lightweight multithreading that extends ideas found in the Cray machines to support larger numbers of threads while reducing the cost of thread management and synchronization.

1 Introduction

In the quest for higher performance, architectures are focusing on multithreading as a way to increase parallelism. In general, programs with highly irregular data and control structures typically found in graph problems are difficult to parallelize efficiently, even though they may contain a great deal of logical parallelism. For Boolean satisfiability solvers (SAT solvers), for example, it has been nearly impossible to obtain significant parallel speedups on conventional SMP systems over serial implementations [6].

SMP systems typically rely on a heavyweight thread model. Heavyweight threads can accumulate a large data trail including a deep call stack, large caches, and significant branch history data. Further, the operations of starting, stopping, suspending or synchronizing heavyweight OS threads incur significant overhead. Because of these factors, heavyweight threads can only be used sparingly within applications to achieve coarse-grain parallelism. By contrast, a lightweight thread has considerably less state associated with it. Unlike heavyweight threads such as Unix pthreads that require OS support, our lightweight threads consist of nothing more than a *frame* of private variables in memory, including a program counter. Lightweight multithreading has its roots in the early work in dataflow and message-passing architectures including P-RISC [10] and Monsoon [12], the J-Machine [11] [4] the Threaded Abstract Machine (TAM) [3], and microservers [1]. The Cray MTA [2] and the upcoming Cray XMT (Eldorado) [7] architectures both rely on lightweight multithreading to attack irregular problems.

As shown in Figure 1, our system architecture consists of a set of compute *nodes* connected by a communication network. Logically, each node contains a portion of global, shared memory and a set of *lightweight processors* or LWPs. A process in the system consists of a collection of *lightweight threads* that communicate through shared memory locations, and each node in the system holds a portion of the threads. The LWPs themselves are completely anonymous, and serve only to process instructions for threads on their node. In place of named registers in an LWP, thread state is packaged in *data frames* of memory. The main difference between a frame and a register set is that frames are logically and physically part of the *memory* system, rather than part of the processor, and that a multithreaded program can have access to many different dynamically-created frames over the course of execution.

Logically, the pool of active threads on a given node is a list of frames in the memory of that node, and operations to spawn or terminate threads simply add or remove frames to or from the list. Whenever a thread performs a memory operation—including spawning a new thread, as well as a

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under its Contract No. NBCH3039003.

1-4244-0910-1/07/$20.00 ©2007 IEEE.

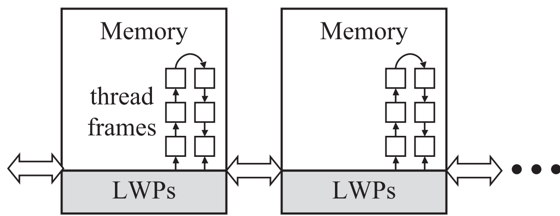


Figure 1. LWP system organization.

load or store—it does so by sending a message to the node that contains a target memory address. All memory transactions are split-phase, and for every parcel request, there is a response that writes a value to the requesting thread's frame. When a thread has all the data it needs to issue the next instruction in its frame, such as the values of operands, that thread is said to be *ready*. Otherwise, the thread is *blocked*. As long as there are enough ready threads so that every LWP can issue an instruction from some thread at every clock cycle, then the system will run at peak performance.

Because threads are part of the memory system rather than the processor state, a process can have orders of magnitude more threads than there are processors, which provides many opportunities for each processor to have an instruction ready to issue. On the other hand, there is a cost to managing lists of threads in memory, and if done in software by the LWPs, the additional instructions required could add substantially to the execution time. To alleviate this burden, we include another processor, called an *ultra-lightweight processor* (ULWP) at the memory bank itself. The ULWP contains minimal circuitry, but yields significant performance improvement for thread management, as well as supporting other atomic operations at the memory.

The goal of this paper is to introduce our hybrid multithreading model, and to provide an initial demonstration of its performance on irregular kernels that are difficult to parallelize efficiently with heavier threads. To accomplish this, we ran an execution-based simulation in a custom environment called SALT with a suite of programs hand-coded in a custom version of C with parallel extensions called DimC. The remainder of this paper is organized as follows. Following this introduction, we first give an overview of the system architecture as it was modeled in the simulation environment. Next, we describe our program kernels and provide the results of the experiments. Finally, we end with conclusions and a discussion of future work.

2 System Architecture

2.1 Structural Organization

Figure 2 illustrates the organization and notional floorplan of a single-chip compute node, called a *lightweight processing chip* (LPC). Each LPC consists of a set of lightweight processors (LWPs), independently-addressable embedded memory blocks (eDRAM), and an intra-node network. Any LWP on an LPC can read or write data to any bank. Each eDRAM bank has an independent controller—the ultra-lighweight processor (ULWP)—which provides hardware support for complex memory accesses including Atomic Memory Operations (AMO), Extended Memory Semantics (EMS), split-phase memory operations, thread frame management and synchronization. In our simulated organization, there are four eDRAM banks for each LWP. Node memory is organized such that each eDRAM bank holds a portion of the pool of ready thread frames for its adjacent LWP.

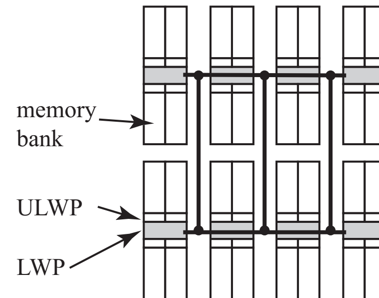


Figure 2. Node functional organization and notional LPC floorplan

2.2 Lightweight Processing and Extended Memory Semantics

Figure 3 illustrates the functional organization of an LWP. In place of a conventional register file, each LWP contains a frame buffer that caches frames for threads that are ready to execute. We determined experimentally that a buffer that holds 128 frames is sufficient. In the actual LWP ISA, a thread's registers *are* the first 32 `Xdwords` in its frame; in other words, the LWP has no general purpose registers in hardware that exist in a namespace separate from memory. In theory the LWP can support unlimited number of threads, where the only limitation on total thread count is the size of memory for storing the threads' frames.

Aside from the frame buffer and a small instruction cache in each LWP, there is no cache in the system and hence no need to maintain cache coherence between nodes. It is important to note that in a massively parallel, lightweight multithreaded architecture, the main role of the cache is to conserve memory bandwidth, rather than to reduce the average memory access time. Moreover, for graph problems that exhibit little locality, as more and more hardware is added to the memory system and as the mem-

ory hierarchy becomes deeper, the memory actually becomes further from the CPU, increasing the latency. In the lightweight multithreading model, the greatest gains are achieved by aggressively pursuing parallelism and keeping the overhead of thread management and context switching low. As long as there are enough threads running in the system (100 threads/LWP), the processor will be at or near peak performance. Therefore, the complex cache-related hardware and protocols can be removed without performance degeneration.

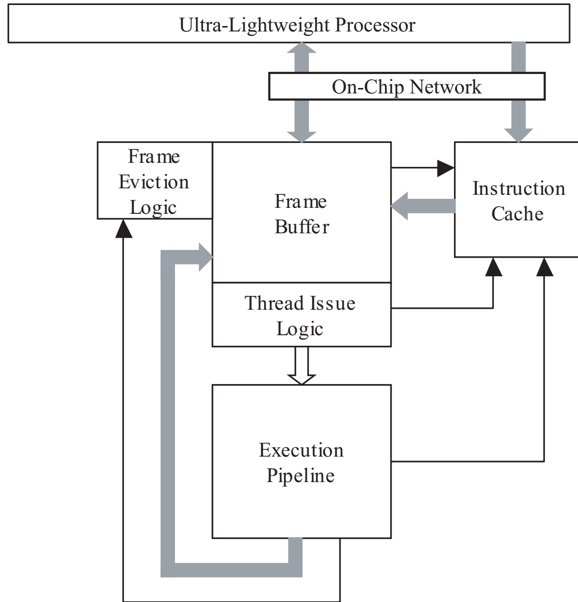


Figure 3. LWP functional organization.

The LWP architecture supports extended memory semantics (EMS) that provide a mechanism for extremely fine grain, lightweight synchronization between threads. The fundamental unit of storage in the architecture is the *extended double word* or `Xdword`. It consists of a 64-bit double word (`dword`) together with a 65th bit called the *extension bit*. By using the extension bit in tandem with mode fields within the `dword` itself, we define a set of extended memory states for a `dword`. If the extension bit indicates the memory is full, the 64- bit field contains the valid data. Otherwise, the 64- bit field contains the metadata needed for hardware to process the memory references. There are a total of 13 defined extended memory states, but for the purposes of this discussion, we can simplify this to 6:

full The `dword` contains a valid value.

empty The `dword` is empty of a value. This typically indicates that a value is pending for the `dword`, and is typically used for producer/consumer synchronization.

FVE forward value leave empty Indicates a store to this dword should be forwarded to the `dword` addressed the metadata and this dword should transition to the E after forwarding data.

FVF forward value leave full Same as the FVE, except the state after forwarding data is F

trap-on-access Invoke a software handler. This situation typically arises when there are multiple readers or writers waiting for access to a location.

error code The cell contains an error code, which may be written as the result of an unsuccessful memory operation, floating point error, or other exceptional event. Typically, an exception is raised when a `dword` containing an error code is accessed.

Because frames are in memory and registers have the same semantics as any memory location, the extended memory state can be used to maintain instruction execution ordering consistency within a thread. Instructions in any given thread issue in order, but may complete out-of-order. An instruction can issue if its input and output registers are available for reading and writing, respectively. In brief, a register is available for reading if it is not empty and it has no pending operations on it. A register is available for writing if it has no pending operations on it. If any of the input or output registers for a given instruction are unavailable, the thread will block when it tries to issue the instruction, to be retried when the input and output dependencies are satisfied.

Under the EMS, load and store instructions are characterized by their *synchronization behavior*. The synchronization behavior describes both the precondition and postcondition of the extended state of the memory location, where the precondition is the state that the memory location must be in before the operation can take place and the postcondition is the state in which the memory location is left after the operation takes place. When a memory operation fails to satisfy the precondition (assuming no other error condition exists), it causes the thread to *block* until the precondition is met. For example, a `load.fe` instruction will block until a memory location is full and then leave it empty, while at `store.ef` instruction will block until a memory location is empty and then leave it full.

Threads block by "queueing up" on the memory location at which they seek to read or write data. The architecture supports this by keeping a pointer to a thread or list of threads blocked on that location. Note that this memory location can be a thread register if it is part of a frame. Figure 4 illustrates the simple case of a synchronization between a single producer and single consumer. Suppose that T2 is a consumer thread that wants to read from memory location

A. When T2 sends a parcel requesting the load and finds address A empty, location A changes state to "forward value, leave empty" (FVE) and its contents is set to the target address of the forward operation, which in this case is register R3 of thread T2. T2 is not placed back on the active list, and is hence blocked on location A. Next, the producer thread T1 sends a parcel to store the contents of its register R2 in A. The system detects that A is in the FVE state, writes the value in R2 to T2 register R3, and leaves A in the empty (E) state. Both threads T1 and T2 are now reinserted on the active lists, unblocking the consumer T2.

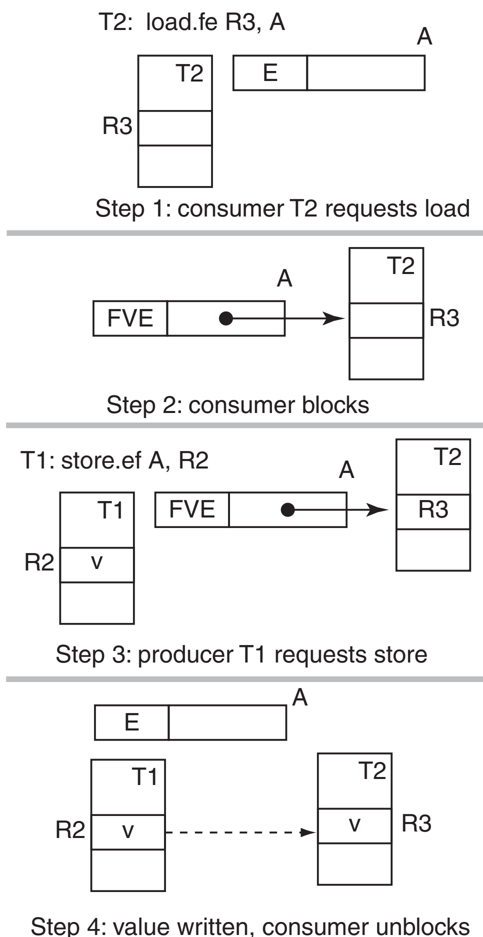


Figure 4. Single producer, single consumer synchronization

While it takes several steps, the common case of a single producer and single consumer can be readily handled by a hardware controller (finite state machine) at the memory bank. For more complex situations such as multiple producers and consumers, software support is needed to manage a queue of blocked threads. To initiate this, memory controller hardware sets a trap on the memory location and creates a special system software thread, named the *EMS handler*, to process the synchronization. If there is heavy contention between threads over a memory location, the overhead of frequently invoking and running handler threads can become very costly. To address this problem, we propose a separate *ultra-lightweight processor* that can run the handler software at the memory bank itself, freeing up the LWP to do "useful" work on application threads.

2.3 Ultra-Lightweight Processing

The ultra-lightweight processor (ULWP) is itself a multithreaded processing engine attached to each memory bank. Because it is designed to interface directly with a memory bank, it can take advantage of wide memory access and have the lowest possible latency. The ULWP supports construction of short programs placed in what normally would be a simple read/write packet being sent by a CPU to a memory. These programs represent relatively short program threads that perform some very specific and localized memory activity. In the ULWP architecture, the short sequence of operations and a very small set of working storage that are performed at a memory is called threadlets [9]. The intent of a threadlet is to represent some short sequence of actions that are to be performed against some very specific memory locations, and which if executed in a conventional design would represent a long latency event. The architecture is designed to minimize the amount of program state in the processor. Figure 5 shows the architecture of ULWP. The main part of ULWP is a 4-stage pipeline. The memory management logic takes care of the execution stage of the memory related instructions such as Load, Store, and Invoke. The memory management logic can also be pipelined to make full use of the burst operation of commodity DRAMs. Unlike the LWP which has 32 general purpose registers, the ULWP only has a few programmer visible register and a program counter of threadlet states. Because the ULWP is a simple design optimized for lightweight transactions at the memory, it is a more efficient way to increase parallelism that by adding more LWPs. Furthermore, since all of the steps in synchronizing a memory operation are handled atomically by the ULWP at the memory without the network-on-chip (NoC) traffic, it minimizes the latency of a complex synchronization operation.

2.4 On-Chip Network and Router Architecture

The network-on-chip (NoC) is processor-memory network for processing parcel requests and replies between LWPs and memory blocks. There is no processor-to-processor communication. The topology used in the simulations for this paper is a mesh, with wormhole routing. Each

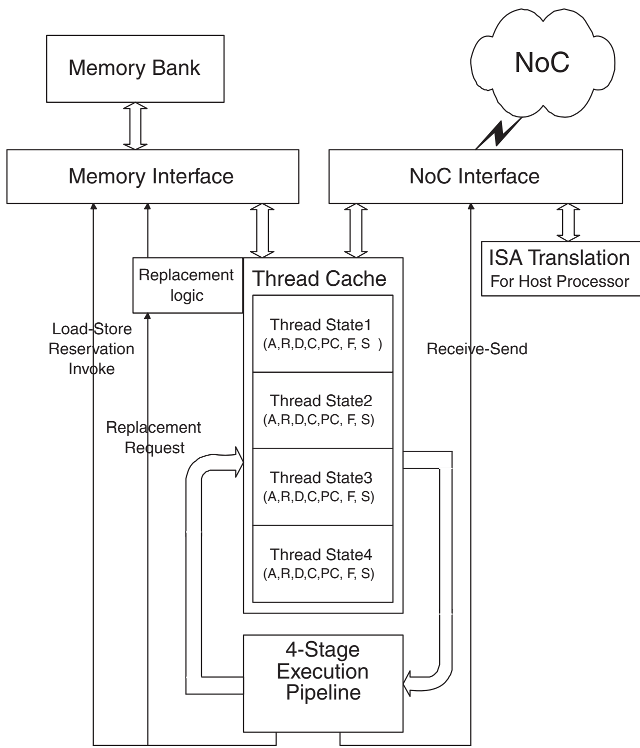


Figure 5. ULWP Microarchitecture

LWP has a dedicated port to one of the routers and each router has connections to each of the neighboring (xplus, xmins, yplus, yminus) routers. Each of the connections between nodes and routers and between the routers consists of two unidirectional links, each with a width equal to flit size. Flits are the units in which the packets are broken down for transmission.

A NoC router consists of four major components: the route computation unit (RC), the virtual channel allocation unit (VA), the switch allocation unit (SA) and the crossbar. In the mesh topology each router has five physical channels: xplus, xminus, yplus, yminus and one for the connection with the local processing element. Each physical channel has multiple virtual channels (VC) associated with it. They are first-in-first-out (FIFO) buffers, which holds the flits from different packets. In our implementation we have used 2 VCs per physical channel. The width of the router link was chosen to be 128 bits. The router architecture differentiates between short parcels (read requests, write acknowledgements) and long parcels (read responses, write request, amo request, spawn request). Short parcels are mapped onto single flits while long parcels are mapped onto multiple flits. In our implementation we use a single stage router to minimize the latency.

3 Experimental Methodology

3.1 Simulation Environment

To evaluate the heterogeneous LWP-ULWP architecture, we developed a custom suite of simulation and programming tools. SALT is a structural-level execution-based simulator that contains a complete implementation of an LPC, with LWP, ULWP, NoC and memory subsystems. The SALT simulation engine is based on the event-driven Enkidu framework, details of which can be found in [14]. To support the thread semantics of the LWP, we developed an extended version of C called, *DimC*. One of the notable features of DimC is that a function call is just a special case of a thread invocation, where the calling thread blocks until the called thread "returns" by synchronizing with it through the standard extended memory semantics. Detailed examples with SALT and DimC can be found in [15].

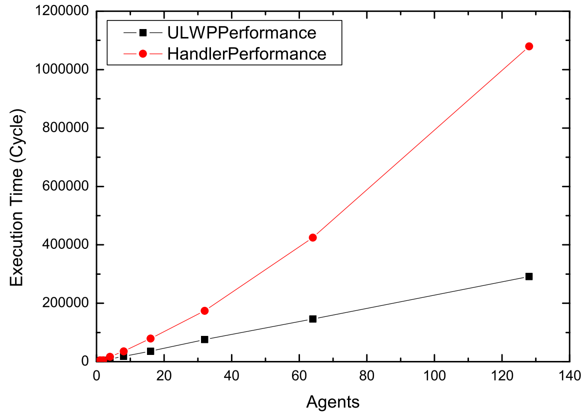
3.2 Benchmark Suite and Results

Because example must be hand-coded for DimC, at this point in the project we are primarily limited to kernel programs and compact applications for evaluation—plans for migrating to a more comprehensive toolchain are underway. To illustrate the capabilities of the architecture, we have chosen four dynamic and irregular problems: competing agents, a SAT solver kernel based on zChaff, N-Queens, and Fibonacci. These benchmarks represent two different categories of irregular problems. The competing agents and the SAT solver have complicated control structures as well as dynamic data structures. They are beyond the common irregular programs. In particular, the SAT solvers program have proven almost impossible to parallelize effectively on the SMPs. N-Queens and Fibonacci are the representatives of the irregular programs with complicated control structures such as recursion. Such programs can achieve decent performance on conventional architectures but need great effort. Each of these kernels and their simulation results are discussed below.

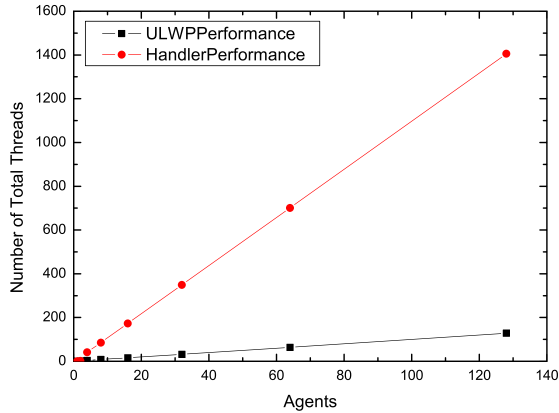
3.2.1 Competing Agents

In this benchmark, multiple competing agents, where each agent is implemented by a single thread, attempt to update a shared memory location simultaneously. While this could be done with atomic memory operations, we have implemented the example using separate synchronizing loads and stores as a vehicle for characterizing the effectiveness of the ULWP in mitigating the cost of synchronization. In each of the experiments, the ULWP runs the system code, which is transparent to the user. Each thread in the experiment attempts to perform 5 updates, for a total of 10 synchronized

loads and stores. The number of competing agents varies from one to 128, and all the agents are uniformly distributed across four LWPs on a single LPC.



(a) Comparison on Execution Time



(b) Comparison on Number of Total Threads

Figure 6. Speedup Comparison for Competing Agents

Figure 6(a) shows a significant difference in the performance of EMS Handler versus that of ULWP. The execution time of synchronization using EMS Handler is 3.7 times longer than that of synchronization using ULWP. When there are few agents, natural latencies in the system prevent updates from different agents all occurring at once, and the memory system can handle the single producer/consumer case without the need to invoke a software handler. As the number of agents increases, requests queue up, requiring management in software. Without the ULWP, the system reaches the point where most of the instructions being issued are from handler threads. Since there are 10 loads and stores per agent, it is easy to predict that the overhead can easily exceed 90 percent, with a small number of LWP cycles spent doing "useful" work.

Figure 6(b) shows a comparison of the total number of threads for this benchmark. Since the agents need to invoke the EMS handler, the more agents we have, the more times the EMS handler needs to be invoked, and the more threads are created. From figure 6(b) when we have 128 agents, there are 1405 threads in total. Only 128 threads are due to agents, and 1277 threads are due to EMS handler. The EMS handler creates 9.9 times as many threads as the agents do. When the ULWP runs the handler, however however, all the synchronization work is done at the memory bank atomically, freeing the LWP to work on the regular application threads.

3.2.2 SAT Solver/zChaff

The Boolean Satisfiability Problem (SAT) is a basic problem from mathematical logic that is fundamental to many problems in automated reasoning, computer-aided design, computer-aided manufacturing, machine vision, database, robotics, integrated circuit design, computer architecture design, and computer network design. In spite of its computational complexity, there is strong demand for high performance SAT-solving algorithms in industry. Over the years, many different approaches and optimizations have been developed to tackle the problem more efficiently. zChaff, the modern variants of the DPLL algorithm [5], has demonstrated state-of-art performance at solving SAT. Recent research, however, has shown that the DPLL based algorithms such as zChaff are extremely hard to parallelize on SMPs, and the performance degeneration for multithreaded implementation on SMPs is significant.

Using lightweight multithreading, there are several opportunities for exposing parallelism in a zChaff-like SAT solver. The main part of zChaff is Boolean constraint propagation (BCP). We use coarse grain, medium grain and fine grain level parallelism to parallelize the program. At the coarse grain level, when a new decision must be made, it is possible to establish two parallel solution trails, one assuming the selected variable is assigned a true value, and one assuming the selected variable is assigned a false value. In the ultimate case, this could expand in a tree-like fashion to have 2N separate and concurrent assignment decision points. Clearly some or even most of them will end up with a contradiction; this needs to be marked in the parent decision point so that the last child coming back with a failure can cause a failure backtrack to the parent.

Medium grain parallelism comes into play as we find unit clauses and generate a new implication to be propagated back through the clauses, all within the same decision step. Each unit clause and the assignment of values to the variables, can be done concurrently. Finally, fine grain par-

allelism occurs when an assignment is to be made to a variable, and we want to (ideally) alert all clauses on the watch list that they need to force an evaluation check from one side, and actually perform this reevaluation. While most of these clause evaluations will end up relinking the clause onto some other variable's watch list, a subset of them will trigger the new implication cases discussed above. Further details on the implementation can be found in [13].

When we apply all three levels of parallelism, the SAT Solver/zChaff benchmark produces a combination of high irregularity and contention. Threads are generated and die dynamically and each of thread communicates with others randomly. Very high contention occurs due to the shared data and communications. Low overhead synchronization is thus necessary to guarantee good performance. From the results of the competing agent benchmark we have learned that the ULWP is more efficient that the EMS handler when doing the synchronization under high contention. Therefore, in this benchmark, all the synchronizations are handled by the ULWP. Because of memory limits of the simulator, only the small data sets were used, which are uniform random-3-SAT data sets from [8], including uf20-91(20 variables, 91 clauses), uf50-218(50 variables, 218 clauses,satisfiable) and uuf50-218(50 variables, 218 clauses,unsatisfiable).

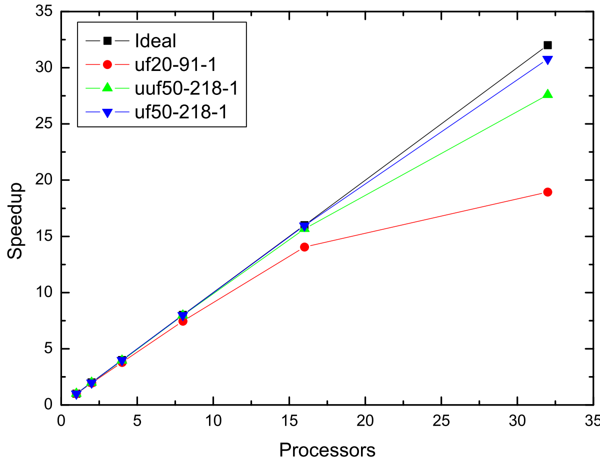


Figure 7. Performance of SAT Solver/zChaff.

Figure 7 shows the performance of SAT Solver/zChaff on hybrid LWP/ULWP architecture. The linear speedup is achieved over a wide range. The saturation is only because of the size of data set. If we can increase the data set, the saturation region will be push to the far right end.

3.2.3 N-Queens and Fibonacci

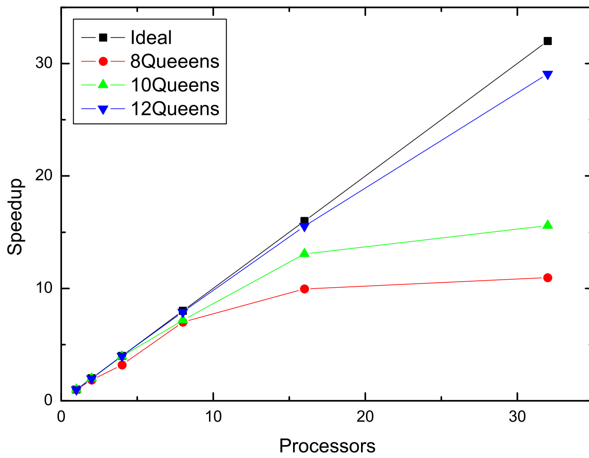
The N-Queens problem is representative of an irregular problem in which it is hard to distribute the workload to different threads, a characteristic found in data flow problem. It seeks to find all solutions to the problem of placing N queens on an NxN chessboard such that no queen can attack another. This is done by recursive calls to the N-queens procedure which evaluates a row of the chessboard based on previous placements.

Figure 8(a) shows the performance for 8, 10, and 12 queens. It is notable this kind of dynamic problem gets very good performance on LWP architecture. In LWP, all the function calls are implemented as threads. As the N-queens program recursively calls itself, new threads are generated. In every clock cycle, new threads are spawned, and some of the old threads die. This dynamic thread behavior is the perfect match for the LWP which has extremely low overhead on thread creation and context switching. The saturations are only due to insufficient workload. It is clear from Figure 8(a) that when increasing the size of data set, linear speedup can be achieved on more LWPs. Although the linear speed can be achieved on conventional architecture, it is much easier on LWP due to architectural features, such as the anonymous threads, threaded function calls and low overhead of thread management and synchronization.

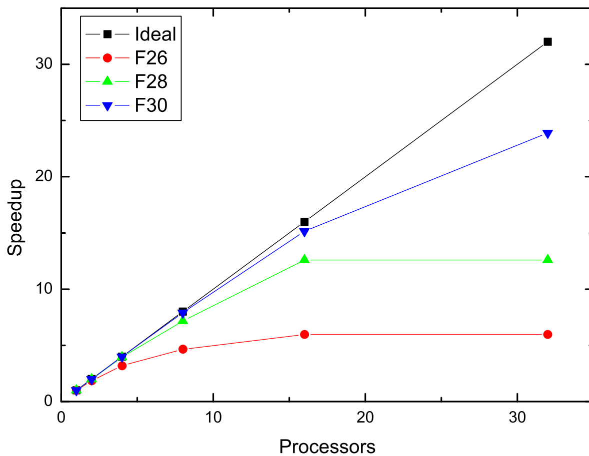
Fibonacci, another important irregular problems with dynamic recursion, computes the nth Fibonacci number based on the well-known equation: Fib(n) = Fib(n-1) + Fib(n-2). For the benchmark, n=26,28 and 30 were used. Figure 8(b) shows that LWP also achieved very good performance. It is can be seen that as long as the size of the data set is big enough, it can always achieve linear speedup on LWP architecture.

4 Conclusions

The heterogeneous architecture of LWP and ULWP has demonstrated the ability to achieve very good performance for the irregular benchmarks which are unsuitable for conventional architectures. One of the main features of our architecture is low overhead thread management, using frames in memory for fast context switches, as well as the ULWP supported extended memory semantics for lightweight synchronization. As noted earlier, the roots of these ideas came from early work in dataflow and hybrid dataflow-RISC architecture. These features become especially important in more dynamic applications where it is impossible to specify the number of threads in advance and new threads are created “on the fly.” As long as an application has sufficient control and/or data parallelism, then a large number of lightweight threads can effectively hide memory latency and achieve near ideal parallel speedups.



(a) Speedup of N-queens



(b) Speedup of Fibonacci

Figure 8. Performance of irregular program on LWP

We note that the compiler we used for our experiments was not very "smart," and we expect that if it had features such as loop unrolling, the number of threads required to avoid processor starvation would have been even smaller, and the performance will be even better. Future work will involve improving the experimental environment so that the large-scale problem can be done on this context.

References

[1] Jay B. Brockman, Peter M. Kogge, Vincent W. Freeh, Shannon K. Kuntz, and Thomas L. Sterling. Microservers: A new memory semantics for massively parallel computing. In *Conference Proceedings of the 1999 International Conference on Supercomputing*, pages 454–463, Rhodes, Greece, June 20–25, 1999. ACM SIGARCH.

[2] Cray Corporation. Cray mta-2 system [online].

[3] David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser, and Thorsten Von Eicken. TAM – A compiler controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, 18(3):347–370, July 1993.

[4] W. J. Dally, J. A. S. Fiske, J. S. Keen, R. A. Lethin, M. D. Noakes, P. R. Nuth, R. E. Davison, and G. A. Fyler. The message-driven processor. *IEEE Micro*, pages 23–39, April 1992.

[5] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[6] Yulik Feldman, Nachum Dershowitz, and Ziyad Hanna. Parallel multithreaded satisfiability solver: Design and implementation. *Electr. Notes Theor. Comput. Sci.*, 128(3):75–90, 2005.

[7] John Feo, David Harper, Simon Kahan, and Petr Konecny. ELDORADO. In *Proceedings of the 2nd conference on Computing Frontiers*, pages 28–34, Ischia, Italy, May 4-6,2005. ACM Press.

[8] Holger H. Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT. pages 283–292.

[9] P. M. Kogge. Architectures for self-contained, mobile, memory programming. U.S. Patent Application 60/411,888, sep 2002.

[10] Rishiyur S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 262–272, June 1989.

[11] M. D. Noakes, D. A. Wallach, and W. J. Dally. The J-machine multicomputer: An architectural evaluation. In Lubomir Bic, editor, *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 224–236, San Diego, CA, May 1993. IEEE Computer Society Press.

[12] Gregory M. Papadopoulos and David E. Culler. Monsoon: An explicit token-store architecture. In *17th International Symposium on Computer Architecture*, number 18(2) in ACM SIGARCH Computer Architecture News, pages 82–91, Seattle, Washington, May 28–31, June 1990.

[13] Lilia Yerosheva Peter M. Kogge. Towards non-copying, highly multi-threaded bcps. Technical report cascade internal technical report, University of Notre Dame, 2006.

[14] Arun Rodrigues. *Programming Future Architectures: Dusty Decks, Memory Walls, and the Speed of Light*. Phd, University of Notre Dame, Notre Dame, IN, 2006.

[15] Srinivas Sridharan. Implementing scalable locks and barriers on large-scale light-weight multithreaded systems. M.s. thesis, University of Notre Dame, July 2006.