# HOW TO EMULATE FINE-GRAINED MULTITHREADING

ANDREAS GRÄVINGHOFF  JÖRG KELLER
FernUniversität-GHS Hagen, Computer Science Dept., 58084 Hagen, Germany

## ABSTRACT

Fine-grained multithreading can be used to hide long-latency operations encountered in parallel computers during remote memory access. Instead of using special processor hardware, the emulation of fine-grained multithreading on standard processor hardware is investigated. While emulation of coarse-grained multithreading is common in modern operating systems, in the fine-grained case research on emulation has been limited and design of multithreaded processors has been favored. It will be shown that latencies encountered in todays parallel computers such as Cray T3E can be hidden by emulation of fine-grained multithreading using a moderate number of threads. Thus, emulation of fine-grained multithreading can be a viable alternative to the expensive design of custom processor hardware with support for multiple threads.

**Keywords:** multithreading, emulation, parallel computers.

## 1 INTRODUCTION

An important problem faced by parallel computers is the latency of accessing remote memory. As this latency usually increases with the number of processors, massively parallel computers are especially affected. While the latency of remote stores can be ignored since they return no result, remote loads have to be completed before computation can proceed (at least beyond a certain point).

A popular approach to attack this problem is to avoid latency by the use of coherent caches. However, this approach causes widely varying memory access times, therefore these machines are called ccNUMA (cache coherent non-uniform memory access) architectures. The non-uniform access time complicates application development, especially for irregular applications. Instead of avoiding latency, one can try to hide latency by *multithreading*, which is explained in the next paragraph.

A multithreaded processor switches context between different *threads* in order to perform useful computations while other threads are waiting for completion of outstanding operations, thus hiding the latency of those operations from the user. This is only possible if several threads per processor are available, hence the application must possess more parallelism than in a ccNUMA machine of comparable size. However, many applications possess this amount of parallelism if the number of threads per processor is moderate.

We distinguish between *fine-grained* (switches context after one or a few instructions) and *coarse-grained* (switches context after a block of instructions) multithreading. In the context of parallel computers, fine-grained multithreading is more interesting (e.g. to ensure synchronous operation of multiple processors as in the PRAM model). Multithreading can be implemented by special processor hardware or by emulation in software on off-the-shelf hardware. Special processor hardware is expensive, time-consuming to design, error-prone and often slower than commercial processors. Examples for multithreaded processors are Sparcle [1], Anaconda [2], SB-PRAM [3] and the Tera MTA [4]. Emulation in software is state-of-the-art in coarse-grained multithreading: it is called *multitasking* and is used by almost every modern operating system. Several new operating systems (e.g. Solaris) support threads in the form of *lightweight processes*. The purpose of these lightweight processes is not latency hiding but reduced overhead compared to normal processes and abstraction from the number of processors available. However, the overhead is still too high for our purposes, hence these lightweight processes are not considered further.

While multithreading is probably the most general approach to latency hiding, it is not the only one. See [5] for an excellent introduction to latency hiding. For example, block transfer or precommunication can be used to hide latency partially. These techniques can supplement multithreading to reduce the number of threads.

We will present the concept of emulating fine-grained multithreading as well as details of an implementation on the Alpha architecture. An evaluation based on this implementation will show that latencies encountered in todays parallel computers such as Cray T3E can be hidden by emulating a moderate number of threads.

## 2 BASIC CONCEPT

The concept of emulating fine-grained multithreading is not new. At least in the case of the SB-PRAM multiprocessor, emulation of multithreading was evaluated as an option [6]. Since the projected performance goals were not met by off-the-shelf hardware in 1991, a custom processor was designed. A description of the SB-PRAM processor, which supports fine-grained multithreading in hardware, can be found in [3].

Emulating fine-grained multithreading on off-the-shelf processors can be done as follows: For each thread, the executed program as well as the *context* (processor state of the thread) are stored in memory. To execute an instruction from a given thread, the emulation program restores the processor state, fetches and executes the instruction and updates the processor state in memory. Afterwards, the next thread is executed. The data structure that contains the context of a thread (and some management information for the emulation program), is called *frame*. To decrease the time to switch contexts, only the part of the context used or modified by the fetched instruction is restored or saved. Note that we assume the emulating and emulated instruction sets to be identical. If this is not the case, then the emulated instructions have to be replaced by one or more instructions of the emulating instruction set.

In the approach by Scheerer [6], the main program contained a subroutine for every instruction type. Based on the fetched instruction word, the emulation program called the appropriate subroutine. Each subroutine determined the required part of the context by examining the instruction word at run-time. Our approach is based on the fact that information about the required context is already available at compile-time.

We modify the program code to replace every instruction $I$ by a subroutine. These subroutines restore every register that will be read or modified by $I$, execute $I$, save every register that has been written or modified, and return. The number and location of registers to be read, written or modified by instruction $I$ can be determined from the instruction set architecture and the instruction itself. This information is used to replace every instruction of the program with the corresponding subroutine. Note that subroutines for instructions of the same type that merely use different operands will be almost identical. After this modification, the emulation program merely calls the subroutines from all threads in a round-robin manner. The program basically consists of a single loop and is sketched below:

1. Initialization (e.g. creation/initialization of frames).
2. Load PC from current frame into register *threadPC*.
3. Execute subroutine by jump to value stored in *threadPC*, saving old PC in register *mainPC*.
4. Return from subroutine to PC stored in *mainPC*, saving old PC in register *threadPC*.
5. Store PC from register *threadPC* to current frame.
6. Load pointer to next frame into register *framePtr*.
7. If the next thread is not the last one, go to step 2.
8. Perform operations that are only necessary once per round (e.g. increasing round counter) and go to step 2.

Since execution of the main loop is required for every instruction, it should be compact and fast. If separation of rounds is not required, steps 7 and 8 can be replaced by a jump to step 2.

We assume that the high-level language source code of the programs to be emulated is available and that the program already uses threads. The thread-related system calls (e.g. creation, deletion) are then replaced by our own routines during recompilation. After modification of the assembler source the program is linked with an additional library containing our routines (e.g. main loop). We further assume that all threads operate in user mode. Thus only the registers accessible in user mode are shared between different threads and form the context of a thread. By confining threads to user mode, the switching time between threads is significantly reduced. Calls to the operating system are not emulated, but are executed as usual by system calls and traps, i.e. there are no changes to the operating system. Obviously, we can not handle self-modifying code, since we perform all code modifications during compilation. However, this is not a serious restriction since self-modifying code is generally considered as unfavorable.

Emulation of multithreading is not restricted to switch context on a per-instruction basis. Instead, context can be switched after a small sequence of instructions from the same thread has been emulated. This will reduce the emulation overhead significantly, since inside the sequence context switches are no longer necessary. We will show later that a context switch requires more clock cycles than the average emulated instruction. Thus, switching context after sequences of multiple instruction will more than halve the emulation overhead. Note that the size of the sequence does not need to be fixed, which makes further optimizations (e.g. identifying sequences of instructions that operate on the same registers and omitting unneccessary accesses to the frame) possible.

## 3 IMPLEMENTATION

We exemplified the emulation of fine-grained multithreading on the Alpha Architecture [7] from Digital. This architecture was chosen for several reasons: First of all, the user mode context contains only data registers, the program counter (`PC`) and the floating-point control register (`FPCR`). For non-floating-point instructions, only the registers explicitly specified in the instruction have to be saved/restored. For floating-point instructions, the `FPCR` has to be saved/restored as well. Second, the large virtual address space simplifies mapping of large shared memories within parallel computers. Third, implementations of the Alpha Architecture (e.g. DECchip 21064, 21164, 21264) continued to be among the most powerful microprocessors available since their introduction in 1992 according to the SPEC benchmarks.

Alpha is a 64 bit load/store RISC architecture. All registers are 64 bit in length: there are 32 integer registers R0-R31 (R31 is read as zero, writes are discarded) as well as 32 floating-point registers F0-F31 (F31 is read as zero, writes are discarded). Instructions are 32 bit in length. Supported data types include longword (32 bit) and quadword (64 bit) integers and five different floating-point formats: VAX F (32 bit), VAX G (64 bit), IEEE single (32 bit), IEEE double (64 bit) and IEEE extended (128 bit). Recent implementations (e.g. DECchip 21264) also support byte (8 bit) and word (16 bit) integers and motional video instructions (MVI). Virtual address space is 43 bit with a physical address space of 40 bit in the case of the DECchip 21164 (21164 for short).

The context of a thread is stored in a data structure (frame) consisting of the following items:

- pointer to the next frame (quadword, offset: 0 bytes)
- number of thread (quadword, offset: 8 bytes)
- status of thread (quadword, offset: 16 bytes)
- floating-point control register (quadword, offset: 24 bytes)
- program counter (quadword, offset: 32 bytes)
- integer registers (28 quadwords, offsets: 40–256 bytes)
- floating-point registers (31 quadwords, offsets: 264–504 bytes)

The integer registers `R28`, `R29` and `R30` are used exclusively by the emulation program, access to these registers by the emulated program is therefore prohibited. As a consequence, storage for these registers is not necessary. This allows us to keep the frame size to be a multiple of the DECchip 21164 cacheline size, which is 32 or 64 bytes depending on cache type and configuration.

Several frames are organized in a ring, with the last element marked with a non-zero status. This data organization allows us to keep the main loop very small:

```
LOOP: LDQ  R29, #32(R30) ;load PC
      JSR  R28, R29      ;execute
      STQ  R29, #32(R30) ;store PC
      LDQ  R30, #0(R30)  ;load frame
      LDQ  R29, #16(R30) ;load status
      BEQ  R29, LOOP     ;check status
```

During emulation, `R30` holds the pointer to the current frame, `R29` and `R28` are used to store *threadPC* and *mainPC* values, respectively.

Based on the instruction type (int, fp, special), number of accessed registers, the need for traps or the floating-point control register, we partition the Alpha architecture instruction set into 19 subsets. Instructions within the same subset use almost identical subroutines. For example, the subset (INT, TRAP, (2/1)) contains all integer (INT) instructions that may cause an exception (TRAP) and use two source as well as one destination register (2/1). These are the the add, subtract and multiply instructions on quad- and longwords with enabled integer overflow. Because of space restrictions, we will only present one basic and one more demanding example of subroutines.

## 3.1 EXAMPLE 1: QUADWORD ADD

The Quadword Add (`ADDQ Ra, Rb, Rc`) instruction uses three registers: The 64 bit sum of register `Ra`'s and `Rb`'s contents is written to register `Rc`. Since the `ADDQ` may cause an arithmetic exception, this instruction is a member of the (INT, TRAP, (2/1)) subset. The subroutine for instructions within this subset is straightforward:

```
LDQ   Ra,40+a(R30) ;load Ra
LDQ   Rb,40+b(R30) ;load Rb
ADDQ  Ra,Rb,Rc     ;instruction
TRAPB              ;trap barrier
STQ   Rc,40+c(R30) ;store Rc
RET   R29,R28      ;return
```

The term `40+a` denotes register `Ra`'s offset within the frame. Register `R30` holds the pointer to the current frame. The trap barrier ensures that arithmetic exceptions caused by the `ADDQ` instruction have been handled before the subroutine returns.

For floating-point additions, i.e. members of the (FP, TRAP, FPCR, (2/1)) subset, the `FPCR` is required to log exceptions and select some rounding modes, which adds some complexity to the appropriate subroutine.

## 3.2 EXAMPLE 2: JUMPS

The Jump to Subroutine (`JSR Ra,(Rb)`) instruction uses two registers: The updated `PC` is written to register `Ra`, and the `PC` is loaded from register `Rb`, ignoring the two lowermost bits. `Ra` and `Rb` may specify the same register. We use arithmetic instructions instead of explicit jumps to modify the `PC`, since this reduces the amount of changes in the instruction stream. To do so, several issues have to be resolved:

- the target address has to point to the subroutine (its first instruction) that replaces the original instruction. This target recalculation can be done during modification of the original program.
- the return address has to point to the next subroutine rather than to the next instruction. This can be done by a simple addition during run-time. The required offset depends on the size of subroutines.

The above reasoning leads to the following result:

```
ADDQ  R29, #20, R27   ;calc returnPC
LDQ   R29,(32+b)(R30) ;load Rb
BIC   R29,#3,R29      ;mask R29
STQ   R27,(32+a)(R30) ;store R27
RET   R31,R28         ;return
```

A minor problem arises if `PC`-relative jumps are emulated. These instructions feature a 21 bit displacement, while integer arithmetic instructions are limited to an 8 bit displacement. Thus the displacement has to be constructed in a separate register, which requires two additional instructions.

## 4 EVALUATION

Our evaluation is based on the Cray T3E massively parallel computer system from Cray Research Inc [8]. The Cray T3E supports between 2 and 2048 processing nodes interconnected by a bidirectional 3D torus. Each processing node contains a 21164 Alpha processor (at 300, 450 or 600 MHz), up to 2 GB of local memory, a router and other supporting circuitry. Instead of an external third-level cache, stream buffers are used to speed up access to local memory via prefetching on previously detected access patterns. Access to remote memory, which is not cached at all, is performed via a large (512 user + 128 system) number of so-called E-registers. A remote load is performed by specifying the target address and an E-register for the result. The result can then be collected from the E-register by a load instruction, which will stall if the result is not yet available. The E-registers significantly increase the number of outstanding loads, since the 21164 itself can only sustain two outstanding loads. The E-registers therefore provide support for hiding the latency of remote memory. For a 2048 processor system, the average and maximum network latency (excluding memory access and processing times) is approximately 1500 and 2500 ns, respectively. At a processor speed of 600 MHz, this translates to 900 and 1500 clock cycles, respectively.

The 21164 Alpha processor is the current implementation of the Alpha Architecture. The 21164 is a superscalar processor featuring two integer as well as two floating-point function units and a sustained (in-order) issue rate of four instructions per clock cycle. On-chip caches include two 8 KB direct-mapped data and instruction caches as well as a 96 KB, 3-way set-associative unified second level cache. Support for an external third-level cache with up to 64 MB is included. Virtual address space is 43 bit large, while only 40 bit are implemented physically. The maximum speed supported by the 21164 is 600 MHz at present.

We will calculate the average number of clock cycles per executed ($\alpha$) and emulated ($\beta$) instruction based on the instruction latencies and issue rules for the 21164 processor. To obtain $\alpha$, $\beta$, we calculate the weighted sum of the instruction latencies presented in Table 1:

$$\alpha = \sum_I \mathrm{w}_I \cdot \mathrm{CPI}_I \qquad \beta = \sum_I \mathrm{w}_I \cdot \widetilde{\mathrm{CPI}}_I$$

The instruction weights are based on measurements for the DLX processor [9, p. C-5]. To obtain the weights, the DLX instructions were mapped to equivalent Alpha instructions and matched to instruction classes as defined in the 21164 hardware reference manual [10]. The results have been scaled and rounded to get a total sum of 1.

The $\mathrm{CPI}_I$ values, where $I$ denotes the instruction class, were taken from the 21164 hardware reference manual [10]. To reflect the superscalarity of the 21164, these

| $I$ | $\mathrm{w}_I$ | $\mathrm{CPI}_I$ | $\alpha$ | $\widetilde{\mathrm{CPI}}_I$ | $\beta$ |
|---|---|---|---|---|---|
| ICOMP | 0.41 | 0.50 | 0.21 | 5 | 2.05 |
| SHIFT | 0.07 | 1.00 | 0.07 | 5 | 0.35 |
| IBR | 0.16 | 1.00 | 0.16 | 4 | 0.64 |
| LD | 0.22 | 1.50 | 0.33 | 7 | 1.54 |
| ST | 0.10 | 1.00 | 0.10 | 4 | 0.40 |
| FADD | 0.02 | 4.00 | 0.08 | 19 | 0.38 |
| FMUL | 0.01 | 4.00 | 0.04 | 19 | 0.19 |
| FDIV | 0.01 | 60.00 | 0.60 | 75 | 0.75 |
| $\sum$ | 1.00 | | 1.59 | | 6.30 |

TAB. 1: AVERAGE NUMBER OF CLOCK CYCLES

values were divided by the number of function units able to perform instructions from this class.

The $\widetilde{\mathrm{CPI}}_I$ values are the maximum number of clock cycles (excluding main loop) spent in any subroutine that corresponds to an instruction from class $I$. The calculation takes into account the 21164 instruction latencies and issue rules. Since the 21164 issues instructions in-order, we can guarantee that all function units are available upon subroutine entry. Note that the `STx` instructions(s) that update the current frame do not issue (due to ressource conflicts) until all previous instructions have been completed.

In addition, several assumptions were made to determine the instruction/subroutine latencies: The (on-chip) first and second level caches are modeled with a miss rate of 7.71 and 8.15 percent, respectively. We computed these values by simulation of address traces for several programs from the SPECint and SPECfp benchmark suites. The simulation was performed using Bryan Hunt's *acs* program [1]. Main memory has a latency of 100 ns as encountered in 21264–based systems using the Tsunami chipset [11]. At 600 MHz, this translates to 60 clock cycles. This latency as well as the first and second level cache miss rates were used to determine the average number of clock cycles per `LD` instruction. We further assume that there are no arithmetic exceptions.

Comparing $\alpha$ and $\beta$, we note that $\beta$ is approximately four times larger than $\alpha$. Along with the latency of the main loop, this constitutes the emulation overhead. Note that a six-fold increase in code size has to be expected. The difference between the size and latency of the subroutines is caused by the inherent instruction-level parallelism.

Several sophisticated models that predict the utilization of a multithreaded processor have appeared in literature [12, 13]. These models take into account the effect of multithreading on other parameters of the system (e.g. caches, network). In contrast, Bianchini and Lim [14] use a simple model of processor utilization and extend it to predict an upper bound on the performance gain achievable by multithreading. We adapt their model to cover emulation of multithreading. The model uses several parameters that are summarized in Table 2: $\alpha$, $\beta$ are the average number of cycles per executed/emulated instruction as calculated above. $p$ is the number of threads, while $C$ is the context switch

[1] acs -a 1 -A 3 -b 32 -B 64 -s 1 -S 1 -i 8096 -d 8096 -U 98304 -x

| | |
|---|---|
| $\alpha, \beta$ | Average no. of cycles per instr. |
| $p$ | Number of threads per processor |
| $C$ | Context switch overhead |
| $\mathrm{t}(1), \mathrm{t}(p)$ | Time between remote loads |
| $\mathrm{T}(1), \mathrm{T}(p)$ | Average latency of remote loads |
| $\mathrm{U}(1), \mathrm{U}(p)$ | Processor utilization |
| $\mathrm{G}(p)$ | Performance gain due to multithreading |

TAB. 2: PARAMETER SUMMARY

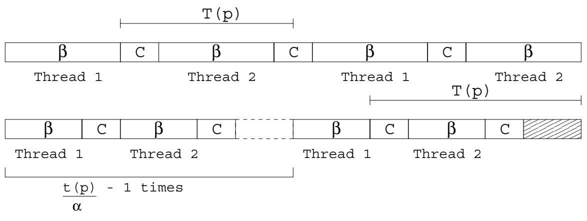

FIG. 1: CALCULATION OF $\mathrm{U}(p)$

overhead (i.e. cycles spent in main loop). Every $\mathrm{t}(1)$ ($\mathrm{t}(p)$ in the multithreaded case) cycles, a remote load occurs. The latency of such a load is $\mathrm{T}(1)$ ($\mathrm{T}(p)$ in the multithreaded case) cycles. $\mathrm{U}(1)$ and $\mathrm{U}(p)$ are the processor utilization in the non-multithreaded and multithreaded case, respectively. $\mathrm{G}(p)$ is an upper bound on the performance gain $\mathrm{U}(p)/\mathrm{U}(1)$ due to emulation of multithreading.

Before we introduce our model, we make several assumptions: Context is switched after every emulated instruction (i.e. after $\beta$ cycles). The time between remote loads $\mathrm{t}(1), \mathrm{t}(p)$ as well as the remote load latency $\mathrm{T}(1), \mathrm{T}(p)$ are constant. The effect of the emulation on synchronization overhead is neglected. We further assume that $\mathrm{t}(1) \geq \mathrm{t}(p)$ and $\mathrm{T}(1) \leq \mathrm{T}(p)$, i.e. multithreading leads to shorter run-lenghts and larger remote latencies due to increased memory traffic and cache pollution [12].

For non-multithreaded processors, utilization $\mathrm{U}(1)$ is given by (1), as observed by Bianchini and Lim [14]:

$$\mathrm{U}(1) = \frac{\mathrm{t}(1)}{\mathrm{t}(1) + \mathrm{T}(1)} \tag{1}$$

To determine the utilization $\mathrm{U}(p)$, we have to cover two different cases as depicted in Fig. 1. In the first case, the latency of remote loads $\mathrm{T}(p)$ can be completely hidden. This is equivalent to

$$\mathrm{T}(p) < ((p-1)\beta + pC) \tag{2}$$

In the second case the number of threads is too small. After $\mathrm{t}(p)/\alpha$ instructions (the number of instructions between remote loads), a remote load occurs that cannot be completely hidden. Since we use $\beta$ instead of $\alpha$ cycles per instruction, we have to scale the utilization with a factor of $\alpha/\beta$ in both cases. The utilization $\mathrm{U}(p)$ is therefore given by

$$\mathrm{U}(p) \quad = \quad \begin{cases} \dfrac{\alpha}{\beta + C} & \text{if (2) holds} \\ \dfrac{p\mathrm{t}(p)}{\dfrac{\mathrm{t}(p)}{\alpha} p(\beta + C) + \delta} & \text{otherwise} \end{cases} \tag{3}$$

where $\delta = \mathrm{T}(p) - ((p-1)\beta + pC)$ represents the amount of latency that could not be hidden. Based on eq.s (1) and (3) as well as the inequalities $\mathrm{t}(1) \geq \mathrm{t}(p)$ and $\mathrm{T}(1) \leq \mathrm{T}(p)$, we derive an upper bound on the performance gain:

$$\mathrm{G}(p) \quad \leq \quad \begin{cases} \dfrac{\alpha}{\beta + C}\left(1 + \dfrac{\mathrm{T}(p)}{\mathrm{t}(p)}\right) & \text{if (2) holds} \\ \dfrac{p(\mathrm{t}(p) + \mathrm{T}(p))}{\dfrac{\mathrm{t}(p)}{\alpha} p(\beta + C) + \delta} & \text{otherwise} \end{cases} \tag{4}$$

We now apply eq. (4) using paramers as encountered in a Cray T3E system. The values of $\alpha$ and $\beta$ were already determined to be 1.59 and 6.30, respectively. The number $C$ of cycles spent in the main loop can be determined in the same way as the $\widetilde{\mathrm{CPI}}_I$ values, which yields $C = 9$. Figure 2 contains four plots of $\mathrm{G}(p)$ according to eq. (4) for $p = 4, 8, 16, 32$ in the range $\mathrm{t}(p) = 2..100$ and $\mathrm{T}(p) = 0..1000$. Note that the graphs intersect the surrounding box at $\mathrm{G}(p) = 1$ instead of $\mathrm{G}(p) = 0$, which simplifies determination of regions that favor emulation of multithreading. As we can see from Fig. 2, small run-lengths and large latencies favor emulation of multithreading. The achievable gain increases with the number of threads. Larger run-lengths require an accompanying increase in latency for the emulation to be competitive.

Now we look at emulation of multithreading on a large Cray T3E configuration by fixing $\mathrm{T}(p)$. We already mentioned the average network latency (excluding memory access and processing times) for a 2048 processor Cray T3E system to be 900 cycles at 600 MHz (processor speed in the T3E-1200 versions). Therefore $\mathrm{T}(p) = 900$. The result is displayed in Fig. 3, which plots $\mathrm{G}(p)$ according to eq. (4) for $p = 4, 8, 16, 32$ dependent on $\mathrm{t}(p)$. Under these conditions, applications with average run-lengths below 70 cycles and utilizing at least four threads can benefit from emulation of multithreading.

Emulation of multithreading favors large systems (i.e. massively parallel computers) with an correspondingly large latency. On the application side, small run-lengths (depending on the number of utilized threads) are required. However, there is no need to emulate a whole application. Instead, emulation can be applied to selected functions that match the required criteria. The system/application domain that may benefit from emulation of multithreading can be increased by reducing the overhead as outlined in Section 2. For example, switching context every two instructions almost halves the value of $C$.

## 5 CONCLUSIONS

We presented a concept to emulate fine-grained multithreading as well as detailed information about an implementation on the Alpha architecture. First, the overhead introduced by emulation of multithreading was determined by calculating the average number of cycles per executed/emulated instruction. The overhead decreases signif-

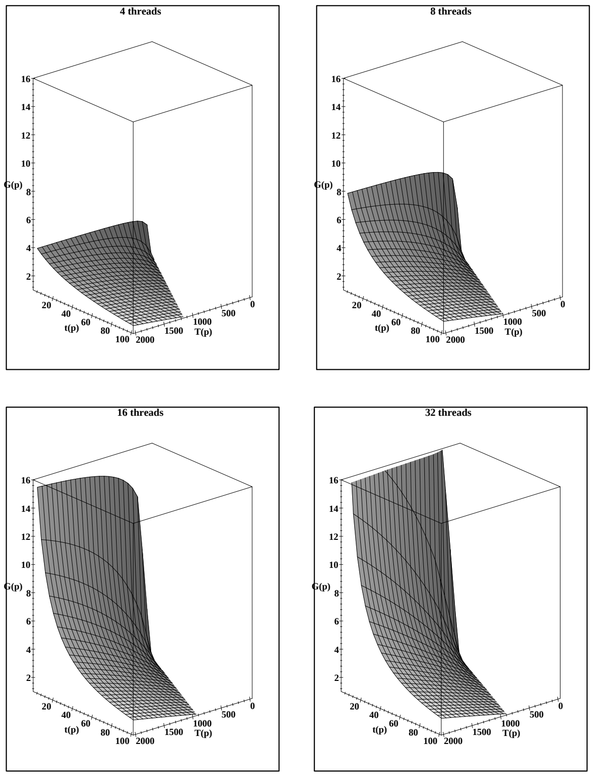

FIG. 2: MAXIMUM GAIN OVER ALL PARAMETERS

icantly by switching context only after execution of several instructions. In addition, the emulation can be performed on several closely coupled processors by splitting the threads between the processors and thus hiding the emulation overhead.

A model for predicting multithreaded processor utilization was used to examine whether the emulation overhead can be offset by the amount of hidden latency. Under conditions as encountered in a large Cray T3E system, characteristics for applications that may benefit from emulation of multithreading were determined.

Our results have yet to be verified by run-time measurements. In order to perform these measurements, access to a 512 processor Cray T3E-900 has been granted by the HLRS (high performance computer center) in Stuttgart, Germany. After implementation of the necessary framework, the concept will be tested on several aplications (e.g. NASPAR parallel benchmark suite).

The current evaluation is based on a processor that issues instructions in-order. The effect of out-of-order execution on emulation of multithreading will be interesting. The next-generation Alpha processor, the 21264, seems to make heavy use of out-of-order execution and register renaming [11] and will therefore be a good platform for measurements in this area.

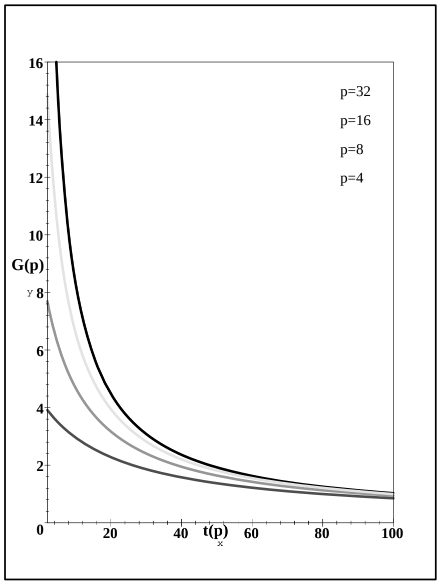

FIG. 3: MAXIMUM GAIN WITH FIXED LATENCY

## REFERENCES

[1] A. Agarwal et.al. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, 13(3):48–61, 1993.

[2] S. W. Moore. *Multithreaded Processor Design*. Kluwer Academic Publishers, Norwell, MA, 1996.

[3] J. Keller, W. J. Paul, and D. Scheerer. Realization of PRAMs: Processor design. In *Proc. 8th Int.l Workshop on Distributed Algorithms*, pages 17–27, 1994.

[4] G. T. Byrd and M. A. Holliday. Multithreaded processor architectures. *IEEE Spectrum*, 32(8):38–46, 1995.

[5] D. Culler and J. P. Singh. *Parallel Computer Architecture*. Morgan Kaufmann, San Mateo, CA, 1998.

[6] D. Scheerer. *Der Prozessor der SB-PRAM*. Dissertation, Universität des Saarlandes, 1995.

[7] Digital Equipment Corporation, Maynard, Massachusetts. *Alpha Architecture Handbook*, 3rd edition, 1996.

[8] S. L. Scott. Synchronization and communication in the T3E multiprocessor. In *Proc. 7th Int.l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 26–36, 1996.

[9] J. L. Hennessy and D. A. Patterson. *Computer Architecture: a quantitative approach*. Morgan Kaufmann, San Mateo, CA, 1990.

[10] Digital Equipment Corporation, Maynard, Massachusetts. *Digital Semiconductor 21164 Alpha Microprocessor Hardware Reference Manual*, 1997.

[11] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, 10(14):11–17, 1996.

[12] A. Agarwal. Performance tradeoffs im multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, 1992.

[13] S. Nemawarkar, R. Govindarajan, G. Gao, and V. Agarwal. Analysis of multithreaded architectures with distributed shared memory. In *Proc. 5th IEEE Symposium on Parallel and Distributed Processing*, pages 114–121, 1993.

[14] R. Bianchini and B. Lim. Evaluating the performance of multithreading and prefetching in multiprocessors. *Journal of Parallel and Distributed Computing*, 37:83–97, 1996.