HPCLatAm 2013, pp. 196-209 (full paper)
Session: Prospective and Ongoing Projects
C. Garcia Garino and M. Printista (Eds.)
Mendoza, Argentina, July 29-30, 2013

Exploring the use of light threads to improve the instruction level parallelism

D. González Márquez[1], A. Cristal Kestelman[2], and E. Mocskos[1]

[1] Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Buenos Aires (C1428EGA), Argentina.
[2] Barcelona Supercomputing Center, Artificial Intelligence Research Institute - CSIC, Barcelona (08034), Spain.

Abstract. One of the main components of any general-purpose machine is the *microprocessor*, this component can be found at the heart of every machine: from standard servers and high performance computing nodes to portable mobile platforms. Its main task is to correctly execute programs as fast as it can, having the production cost and consumption as border conditions.
The research in processor architectures centers in optimizing the processor design according to the specified functionality and having into account present and future technologies. This optimization can be based on different parameters: performance, consumption, production cost, surface.
In this work, we propose a novel mechanism combining software and hardware that allows to improve the Instruction Level Parallelism using simple cores and light threads. A modified processor is implemented using a simulation tool and two examples are presented: sorting an array and filtering a matrix. In both cases, promising results are obtained.

1 Introduction

One of the main components of any general-purpose machine is the *microprocessor*, this component can be found at the heart of every machine: from standard servers and high performance computing nodes to portable mobile platforms like smartphones. Its main task is to correctly execute programs as fast as it can, having the production cost and consumption as border conditions.

The research in processor architectures centers in optimizing the processor design according to the specified functionality and having into account present and future technologies. This optimization can be based on different parameters: performance, consumption, production cost, surface.

Currently, due to technological limitations, the high performance computing processors are multi-core and multi-threaded, which diverted heavy research activity towards this field. Parallelism allows the hardware to accelerate applications by executing multiple, independent operations concurrently [6]. Parallelism can be found at three levels: instruction-level parallelism (ILP), thread-level parallelism (TLP), and data-level parallelism (DLP). Complex techniques such as

multiple instruction issue, out-of-order execution, speculation, and aggressive branch prediction were used at extremely high clock rates, which resulted in i) increased design complexity, ii) under-utilized silicon area, and iii) excessive power dissipation [12].

Improvements in two of the three factors that have historically driven processor performance ILP and gates per clock, have essentially reached their limit [7]. The third factor, process technology, is also driving architectural change as it becomes wire and power limited rather than device limited. Future architectures will employ explicit parallelism to compensate for flat ILP to allow to continue increasing the performance, feature modularity to minimize the use of global wires, and exploit locality and heterogeneous architectures for power efficiency. Fortunately most emerging applications have large amounts of explicit task and data-level parallelism and can be mapped to these new architectures. Unfortunately, even in these new machines, power constraints limit achievable performance [2].

Typically, a *process* consists of CPU state, kernel stack, working directory, open file descriptors, signal table, signal mask, user id, group id, memory map. The CPU state includes a stack pointer (SP), which points to the current activation frame on the stack, the program counter (PC), which references the current instruction; and the remaining registers that store other global and local data [11]. Switching from one process to another introduces a large overhead. The CPU state and the other process information must be stored, the Translation lookaside buffer (TLB) is typically flushed and reloaded, and the next scheduled process must restore its own CPU state and related information before starting its execution. Communications between processes could be a complex procedure.

A thread is a light-weight process developed to overcome many of these shortcomings. Multiple threads can coexist within a process and share its memory map, file descriptors, code, and global data. The state of each threads is only composed by a program counter, stack pointer, stack, general purpose registers, and a small amount of additional thread management information. The adoption of user threads (threads directly controlled by the user) is a way to avoid excessive user-kernel boundary crossings, which should be minimized in order to achieve good performance, specially when using tens or hundreds of thousands of threads.

Creating and running a single thread introduces overhead. Overheads include thread creation, deallocation and any scheduling costs that may be incurred when running a thread [11].

Current high-end commodity processors support eight or more simultaneous threads of execution per CPU socket. Most computational nodes and scientific workstations contain several multi-core sockets, allowing them to deliver sixteen or more concurrent threads of execution. Multi-threaded applications that fully utilize this hardware capability can notably improve their performance. A multi-threaded application breaks an application into multiple pieces, or threads of execution, that run concurrently. Applications that use one or very few threads of execution will likely not benefit from newer architectures [3].

Software development that do not take into account the new multi-core technologies will end with a single-threaded and poorly scaling software, not able to take advantage of the extra processor cores. OpenMP is a framework focused on scientific computing aim to help developing application using multiple threads. Fürlinger et. al. analyzed the scalability behavior and the overheads of OpenMP applications [4]. Analyzing and understanding the scalability behavior of applications is an important step in the development process of scientific software. Inefficiencies that are not significant at low processor counts may play an important role when more processors are used and may limit the application's scalability.

The task of dividing programs into threads that will be executed in parallel is rather straight forward for regular or numeric applications. Despite of introduced overheads, the current compiler technology can perform it efficiently if enough work is present. However, for general purpose programs (i.e. non-numerical programs), compilers usually fail to discover the potential thread-level parallelism that could be effectively exploited [10].

In current multiprocessor systems, the processors are treated as independent functional units. In this work, we propose a novel mechanism combining software and hardware support to further improve ILP using light threads. This approach is based on the use and organization of simple processors (i.e. in-order execution) to decrease their complexity and improve power consumption. Out-of-order execution consumes a significant amount of power by design[13]. The issue queue together with the wake-up and select logic stands for a significant fraction of the total energy consumed by a processor. Reported numbers are in the range from 18% for the Alpha 21264 [5], 19% for the Pentium Pro [9], and up to 40% for the Pentium 4 [8].

The main idea is to consider a set of cores as an individual processor. The work is assigned to this processor and the related execution threads are managed and dispatched to the *internal* cores. The programmer is in charge of controlling the available resources (i.e. the internal cores) using the support of the language, compiler, and operating system.

This work tackles the problem of running very small chunks of code in parallel. Executing these tiny parallel pieces of code would be inefficient due to the involved overhead using current multi-thread programming tools (like `pthreads` and `openMP`).

The rest of this work is organized as follows: in section 2 the general architecture is shown, also some use examples are include. The section 3 the details of the experiments are given and the obtained results are discussed. Finally, we draw some conclusions and review some possible lines of future work in section 5.

2 Architecture

The large overheads related with thread creation, deallocation and scheduling impose a threshold to the achievable parallelism. The key concept of this pro-

posal is reducing these costs in a scenario of multiple threads that are executing in multiple processors: a mechanism for administrating the execution contexts related to the same memory map, same process and, even to the same memory cache.

The new hardware is designed in order to work similarly to `pthreads`. The idea is to run small pieces of code in different processors. The challenge is to treat with the overhead introduced by the creation and starting of the threads: if the amount of useful work is not enough, the proposed mechanism would turn useless.

If we manage to reduce the overhead introduced by creating a thread using hardware support, this should allow running very small pieces of code regardless the operating system. The hardware support should have instructions to allow configuration, starting and assigning a new thread from a processor to another one.

The proposed architecture adds three important changes at hardware level:

1. Store an execution context: for a given task, we need to store the context, and be able to execute it in the available cores.
2. Detection of task ending: need to know when a given task finished. A new synchronization mechanism is added, this allows to stop and restart cores.
3. Modification to the instruction set: add new specific instructions to start, stop and synchronize threads and cores.
   The new instructions specifically added to manage the threads are:
   - `mth_run`: starts a new thread in a free core
   - `mth_delegate`: pause a thread and put the last one on the waiting queue
   - `mth_end`: stop the current thread
   - `mth_syn`: wait until the end of other thread

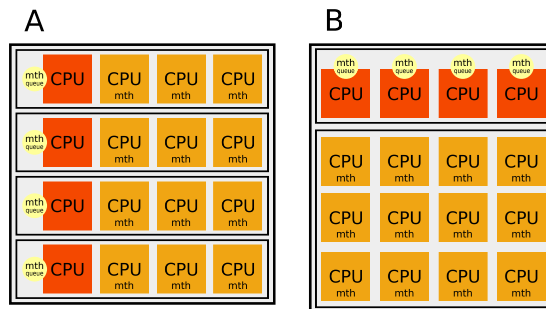


Fig. 1. Two possible processor internal organization: (A) each processor has a predefined set of associated cores in which it can start light threads or micro-threads, (B) each processor can start a thread in any available processor in a pool.

In this architecture, there is no information about which processor should be used or about the task's characteristics. This proposal is based on a *task queue*. In the case of no available cores for running a new task, the corresponding context

is stored in a special structure, the *queue*. This is exemplified in the figure 1 in which this queue is incorporated to the red processor. When a core becomes ready, the execution context will be delivered to it from the queue, its state is load and the thread is started. This design allows to have severals threads *in fly*. The synchronization between threads is implemented using an specific instruction to set a core in an active waiting state.

The figure 1 shows two possible organizations for the multi-core processor. There are two type of cores: i) *full cores*, which control the execution context queue (red cores in the figure) and ii) *mth cores* which are simpler cores that execute a light thread or micro-thread (mth).

The main difference between the mentioned organizations is:

- (A): there are a predefined set of mth cores assigned to each full core.
- (B): each full core can control (i.e. send micro threads) to any mth core.

The operating system only considers the the full cores to be scheduled and managed, any type of signal or exception is handled by the full core, even if it is generated by a mth core. The operating system delegates the mth core management and scheduling to a full core, although it can access the mth core internal state and restart it if necessary.

The internal organization of the (A) case can be used selecting a full core and a set of mth cores to a specified task or application, giving to this application the control of which mth cores are used. The associated limitation is that the application parallelism is bounded by the number of mth cores assigned to the full core, amount which is fixed by the processor architecture. Even though this could be taken as a severe restriction, the adopted programming model could not use a large or variable number of mth cores. Furthermore, this internal organization allows the implementation of an independent communication channel between the mth and full cores.

The case (B) can be fitted to a programming model requiring a large and variable number of mth cores in the applications. In this scenario, a pool of mth cores is introduced, each one can be managed by any full core. Two initial problems are easily to point:

1. Having a pool of mth cores requires sharing a communication channel between all the cores, which means adding complexity to the processor organization.
2. Each application in the system could have a dynamic number of mths, each one assigned to potentially different full core. The communication between the threads should be considered due to the need of sending the process execution context from one core to another. If this communication has to be frequently done, adding an independent channel to each full core can help alleviate this load.

These considerations show the importance and the coupling between the programming model, the operating system and the proposed architecture.

```
mth_run

  IF there is free core
    core(free).state <- core(0).state
    core(free).state.register(0) <- 0
    core(free).state.pc().next_inst()
  ELSE
    new store_state
    store_state.state <- core(0).state
    store_state.register(0) <- 0
    store_state.pc().next_inst()
    mth_queue.add_back(store_state)
  FI
  core(0).state.register(0) = new mth_thread_id
```

```
mth_end

  IF mth_queue.empty()
    core(current).halt()
  ELSE
    core(current) <- mth_queue.get_first_state()
  FI
```

```
mth_delegate

  IF mth_queue.not_empty()
    new store_state
    store_state.state <- core(current).state
    mth_queue.add_back(store_state)
    core(current) <- mth_queue.get_first_state()
  FI
```

```
mth_syn

    WHILE task_not_end(mth_thread_id) DO
    OD
```

Fig. 2. Pseudo-code executed by the specific instructions implementing the `MTH` mechanism. `core` is a processor array and `mth_queue` is the pending micro-threads to execute.

Code Example

In this simple example, we use a version for the light thread mechanism (`MTH`) based on a *Alpha* microprocessor due to its simplicity of the implementation. We apply three of the four new instructions: i) `mth_run` for MTH creation, ii) `mth_end` for MTH finalization, and iii) `mth_wait` for setting the MTH in waiting state.

The figure 3 exemplifies the execution of a thread in a processor and the steps needing to spawn a new thread:

A) The main thread stores two data in the registers (`data_1` and `data_2`), these registers will be used as input parameters for the function to be executed.
B) The main thread modifies the stack pointer, pointing to a new (empty) stack.
C) A free core is selected, the context is copied from the core running the main thread to the new one (named Core mth in the figure). The context of the two processors differs in the value stored in register 0, which was accordingly changed in each one. This register is one of the register bank, any could be used depending of the program.
D) The main thread continues its execution and jumps to the label `jmp` where the original stack is restored.
E) The recently created MTH compares the register 0 and executes the other branch. As can be seen, this mechanism can be used to divide the execution flows between the recently created MTH and the main one.
F) After some processing, the main thread finishes and starts waiting for the other thread.
G) When the new MTH finishes, the execution of `mth_end` instruction activates the main thread.

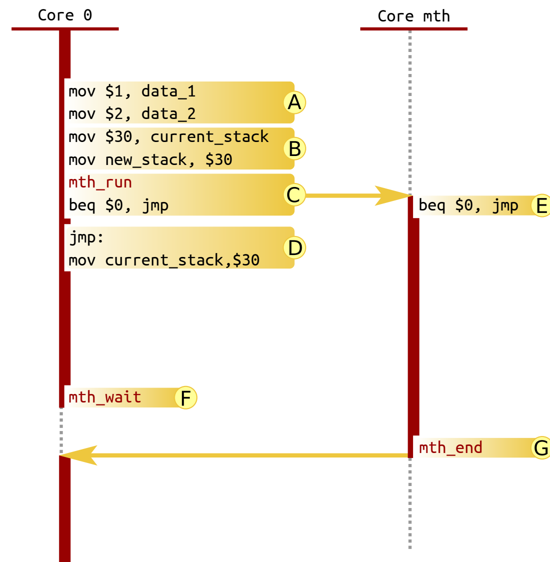


Fig. 3. MTH mechanism at work. (A) The main thread stores two data in the registers (used as input parameters for the function to be executed), (B) Then, modifies the stack pointer (a new empty stack is selected), (C) The context is copied from the core running the main thread to the new one, The register 0 has a different value in each thread, (D) The main thread jumps to the label `jmp`, the original stack pointer is restored, (E) The recently created MTH does not branch (because the value of register 0), (F) The main thread finishes and starts waiting for the other thread, (G) When the new MTH finishes, `mth_end` instruction activates the main thread.

This procedure shows the interaction between threads and the conditions needed by `MTH` mechanism to be efficient. The setup procedure for the new thread and the call to `mth_run` instruction should be done fast enough to allow small pieces of code to be delegated to other cores.

Synchronization

Synchronization between threads is implemented using a special instruction (`mth_syn`). This instruction produces the suspension of the calling thread until another core finishes (i.e. call `mth_end`). The active synchronization is used based on the assumption that each thread has a similar amount of assigned work and, in the case of needing to wait, they have to do it for few clock cycles. If the need for longer waiting periods raises, another synchronization mechanism should be considered, for example enqueue the current thread instead of executing a busy waiting. This waiting mechanism is only used by the main thread, the rest of the spawned threads never have to wait.

Programing Model

The proposed mechanism (micro-threads -MTH-) follows the pthreads programming model, based on the fork-join mechanism. The main difference lays in the amount of work (i.e. clock cycles) needed to execute a fork: in MTH is significantly lower than in pthreads.

The thread creation is controlled by the programmer. This procedure is shown in the figure 3 exemplifying the setup and starting of a new thread.

When a thread is created, the mth_run instruction copies the needed parts of the execution context from the original core to the new one (i.e. the core selected by the MTH hardware mechanism to run the new micro-thread). In the case of no available processor, this instruction copies the context to a waiting queue. Once a processor becomes free, this context is copied from the queue to the processor and the new micro-thread is started.

Both threads (the original and the new one) will execute the same sequence of instructions, one selected register is used to differ both threads (for example, register 0). In this way, both threads can select a different branch path using this register value, and execute a different sequence of instructions. In the figure 3, this is shown at point (C) and (E): both threads chooses a different path of the branch based on a different value stored in register 0. As a remark, the programmer (or compiler) can decide if both threads executes different codes, as the use of the branch is not mandatory.

One important aspect of this model is that the micro-threads lives at user space, the operating system interacts only with the main thread (i.e. all the generated interruptions and exceptions are handled by the main thread).

3 Methodology and Results

In this section, the experimental details for testing the MTH mechanism are described. The processor used in the simulations is based on the in-order ALPHA architecture, supporting its full instruction set. The in-order processor allows to use the simulated ticks as a measure of the time consumed by each function.

The processor supporting the MTH mechanism is simulated using GEM5 tool [1]. This tool is modified by adding a new module for managing micro-threads: they are treated as execution contexts that can be executed in any available processor in the system. This module is implemented at the system level in GEM5 and its main functions are: i) saving the execution contexts to be executed in the cores, ii) tracing the processor executing each micro-thread and iii) storing the context waiting queue. Additionally, the module implements the specific instructions introduced to support the MTH mechanism: mth_run, mth_delegate, mth_end, mth_syn.

For example, this module has the responsibility to launch a new micro-thread every time a processor becomes available by accessing the queue, and to access the processor file register and internal registers to update them with the execution context.

The two implemented examples used to test the MTH mechanism are based on the pthreads philosophy to solve problems, but attacking very small instance sizes to solve. The first problem is solved using a divide & conquer methodology, parallel executing each part. In the second example, the problem is divided without modifying the code and only identifying independent fragments of code, resembling a standard compiler way of work.

Example 1: Sorting

The first implemented example consists in ordering a random set of elements stored in an array.

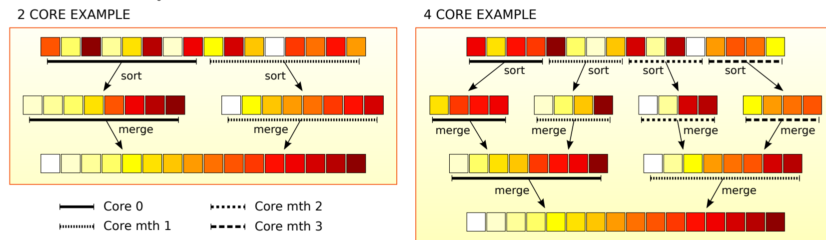


Fig. 4. Sorting strategy with mth: the disordered array is split in parts (same number as available cores). Each part is sorted using a heap-sort in an available processor. The partially ordered parts are then merged.

In this example, heap-sort is initially used and, after obtaining the ordered pieces of the array, a merging procedure is implemented.

The first step in the procedure is splitting the disordered array in as many parts as available processors (in this example, 2 and 4 parts). The initial slices are sorted using heap-sort, then the ordered pieces are merged. The figure 4 shows this procedure. The example for two cores is treated in the left side of the figure: the disordered array is split in two parts, each one is assigned to an available core. These disordered parts are sorted using a heap-sort algorithm, then the two ordered parts are merged in the core 0. The example for 4 cores is similar to the previous one, except that a new merging stage is added: the first one merges the four ordered parts in two new partially ordered ones, and the second stage obtains the final ordered array.

Figure 5 shows the results obtained for the sorting procedure considering three different instance sizes: 32, 64 and 100 elements. For two cores, the obtained results show interesting speed-ups: 1.6 times for 32 elements and 1.8 times for 64 and 100 elements (compared against single core). In all cases, both cores present a fair load balance.

When solving the problem with four cores, the main difference is the emergence of an unbalanced load: cores 2 and 3 have less assigned work than cores 0 and 1. As is shown in figure 4, the second stage of the merging procedure is assigned to cores 0 and 1, leaving cores 2 and 3 in idle state.

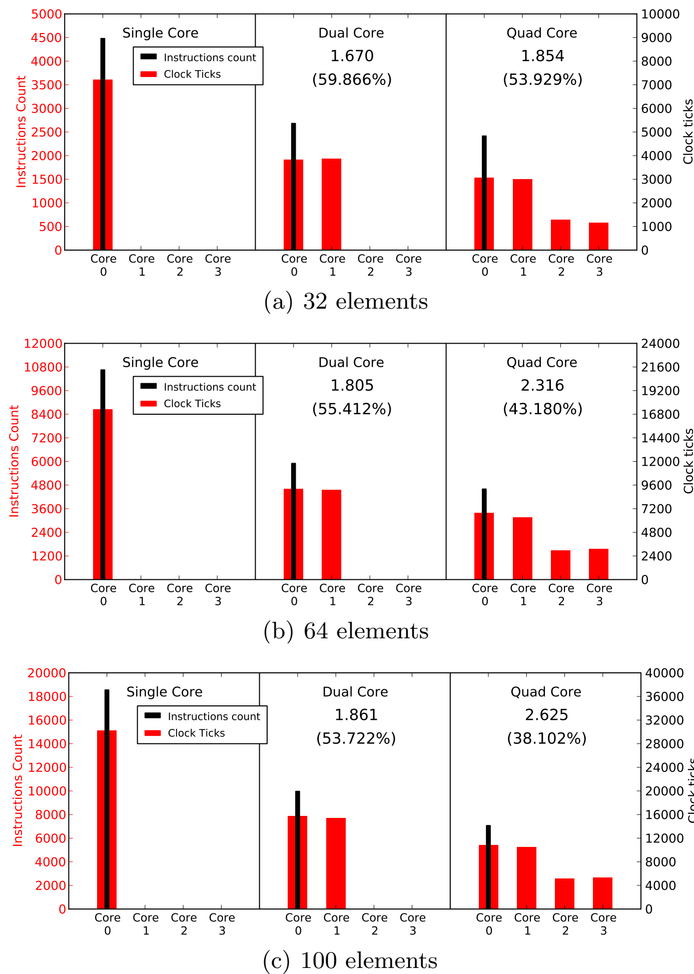


Fig. 5. Using `MTH` mechanism to sort an array of 32, 64 and 100 elements. The unbalanced load in the four cores case comes from the implemented algorithm: two cores remain idle while the other two compute the last stage of the merge.

Example 2: Filter

In this example, the procedure consists in the implementation of a filter based on a *threshold* value. A matrix is traversed and each value is kept if it is greater than the selected threshold.

The simplicity in the parallelization methodology lays in the fact that each value can be computed independently from the others, this is usually known as an embarrassingly parallel problem. In this case, the only difficulty to tackle is having enough work to do in each parallel worker.

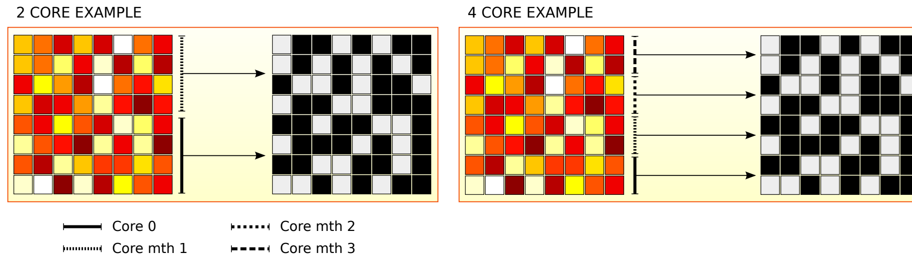


Fig. 6. Threshold based filtering using MTH mechanism: if the inspected value is less than a selected value (threshold), it is painted *black*. The matrix is split between the present cores in the system. Each matrix value can be independently processed.

The figure 6 shows the implementation of the algorithm: the matrix is split in as many parts as available cores. Each micro-thread inspects each value and paint it *black* if the condition is not met (being greater than the threshold value). Following the same methodology than in the previous example, running the algorithm with two and four cores are considered.

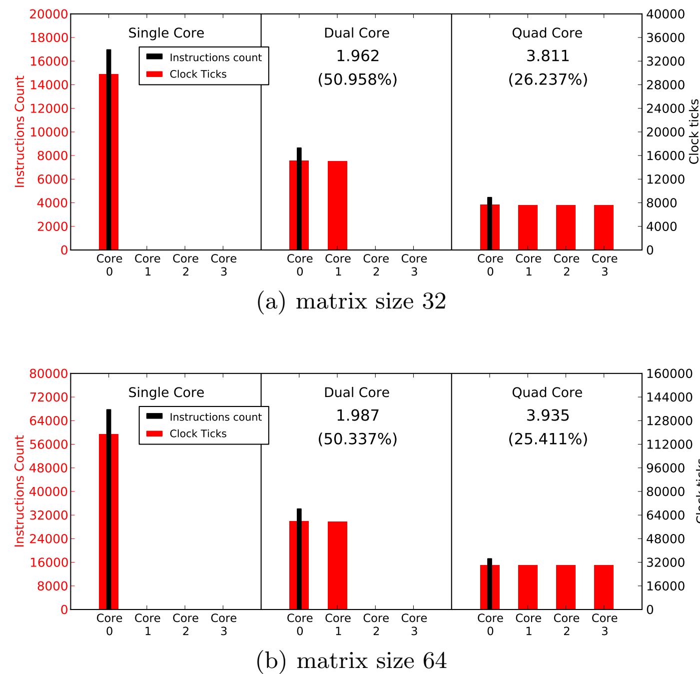
Figure 7 shows the obtained results for two problem sizes: matrix with 32 and 64 elements. The obtained results show a near the optimum performance. The reported speedup for two cores are near two times respect single core for both problem sizes. The behavior in the case of four cores is very similar, presenting a speedup of almost four times.

The independence in the operations of the implemented algorithm allows to obtain these very interesting results. Although similar results could be expected if the same problem is solved using `OpenMP`, the `MTH` mechanism allows treating efficiently very small instance problems.

4 Discussion and Future Work

Some points in this work should be marked. Copying a micro-thread execution context is an operation that necessarily should take some time, but could be handled using an overlapping mechanism. For example, when a processor is near to start the execution of a `mth_run` instruction, the new core can be started in advanced with the available information already copied to the corresponding structures. Even though, if the initial instructions in the spawned micro-thread do not use the execution context values, they can be increasingly copied when they are needed by the processor. Another point where some optimization can be included is before the execution of the `mth_end` instruction. The system can prepare the next micro-thread to be executed and send it to the processor while it is still executing the previous micro-thread.

In the considered examples, all the needed processors were available when needed, but in more realistic examples, this situation will not be the usual one.



(a) matrix size 32

(b) matrix size 64

Fig. 7. Using MTH mechanism to filter a matrix of 32 and 64 elements. Almost optimum speedups are obtained due to the embarrassingly parallel nature of the problem.

The access to the memory is limited, increasing the amount of present cores in the system will stress the memory system. The limit will be impose by the available cores sharing the access to the memory and the size of the problems to solve.

In the presented examples, the tasks are very simple and can be highly parallel if the overhead do not make it unproductive. The usual parallelization tools (like `OpenMP`) relies in having enough work to do: the gain in computing time should overcome the overhead of scheduling a new thread. The proposed `MTH` mechanism showed that can handle smaller size problems.

The included examples only show the potential of this proposal. Exists two additional considerations to solve in future works: i) interaction between several processors (each one having associated mth cores) and ii) the behavior of the operating system respect controlling the processors and their associated mth cores.

Usually, the parallel code includes the generation of new threads inside a do-while cycle: in every iteration a new thread is scheduled. In this type of applications, new threads have to be managed and synchronized dynamically

during the most part of the application execution. To deal with this important class of parallelism, the model should be enhanced with new functionalities.

5 Conclusions

One of the main components of any general-purpose machine is the *microprocessor*, this component can be found at the heart of every machine: from standard servers and high performance computing nodes to portable mobile platforms. Improvements in two of the three factors that have historically driven processor performance ILP and gates per clock, have essentially reached their limit. The third factor, process technology, is also driving architectural change as it becomes wire and power limited rather than device limited.

Creating and running a single thread introduces overhead (i.e. creating, running, deallocation and scheduling). Current high-end commodity processors support eight or more simultaneous threads of execution per CPU socket.

The proposal introduced in this work is designed in order to work similarly to pthreads: running small pieces of code in different simple processors. The proposed architecture adds three important changes at hardware level: i) Store an execution context, ii) Detection of task end, and iii) New specific instructions to start, stop and synchronize threads and cores.

Two implemented examples were shown: one easy to parallelize (i.e. threshold filter) and one needing a merging procedure which difficult the parallelization (i.e. sorting problem). The obtained results show that the `MTH` mechanism can be used to treat small problem instances using 2 and 4 simple cores with low added overhead. For all the considered examples and number of cores, the performance increase with the problem size.

Acknowledgments

E. Mocskos. is researcher of the CONICET (Argentina). D. González Márquez has a scholarship from CONICET (Argentina). This work was partially supported by the cooperation agreement between the Barcelona Supercomputing Center and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contracts TIN2007-60625 and TIN2008- 02055-E; by the European Community Seventh Framework Programme [FP7/2007-2013] under the ParaDIME Project, grant agreement no. 318693 and RISC (288883); Universidad de Buenos Aires (UBACyT 20020100100889) and CONICET (PIP 1087/09).

References

[1] Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A.: The gem5 simulator. SIGARCH Comput. Archit. News 39(2), 1–7 (Aug 2011), http://doi.acm.org/10.1145/2024716.2024718

[2] Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., Burger, D.: Dark silicon and the end of multicore scaling. In: Proceedings of the 38th annual international symposium on Computer architecture. pp. 365–376. ISCA '11, ACM, New York, NY, USA (2011)

[3] Farber, R.M.: Topical perspective on massive threading and parallelism. Journal of Molecular Graphics and Modelling 30, 82–89 (2011)

[4] Fürlinger, K., Gerndt, M.: Analyzing overheads and scalability characteristics of openmp applications. In: Proceedings of the 7th international conference on High performance computing for computational science. pp. 39–51. VECPAR'06, Springer-Verlag, Berlin, Heidelberg (2007), http://dl.acm.org/citation.cfm?id=1761728.1761733

[5] Gowan, M., Biro, L., Jackson, D.: Power considerations in the design of the alpha 21264 microprocessor. In: 35th annual conference on Design Automation, 1998. Proceedings. pp. 726–731 (1998)

[6] Hennessy, J.L., Goldberg, D., Patterson, D.A.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, 5th edn. (Sep 2011)

[7] Horowitz, M., Dally, W.: How scaling will change processor architecture. In: Solid-State Circuits Conference, 2004. Digest of Technical Papers. ISSCC. 2004 IEEE International. pp. 132–133. No. 1 (2004)

[8] Hsiao, K.S., Chen, C.H.: An efficient wakeup design for energy reduction in high-performance superscalar processors. In: Proceedings of the 2nd conference on Computing frontiers. pp. 353–360. CF '05, ACM, New York, NY, USA (2005)

[9] Manne, S., Klauser, A., Grunwald, D.: Pipeline gating: speculation control for energy reduction. In: ISCA '98: 25th Annual International Symposium on Computer Architecture, 1998. Proceedings. pp. 132–141 (1998)

[10] Marcuello, P., Gonzalez, A.: Thread-spawning schemes for speculative multithreading. In: High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on. pp. 55–64 (2002)

[11] Price, G.W., Lowenthal, D.K.: A comparative analysis of fine-grain threads packages. Journal of Parallel and Distributed Computing 63, 1050–1063 (2000)

[12] Soliman, M.I.: Design, implementation, and evaluation of a low-complexity vector-core for executing scalar/vector instructions. Journal of Parallel and Distributed Computing 73(6), 836 – 850 (2013), http://www.sciencedirect.com/science/article/pii/S0743731513000282

[13] Vandierendonck, H., Manet, P., Delavallee, T., Loiselle, I., Legat, J.D.: By-passing the out-of-order execution pipeline to increase energy-efficiency. In: Proceedings of the 4th international conference on Computing frontiers. pp. 97–104. CF '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1242531.1242548