

Analysis of Several Scheduling Algorithms under the Nano-Threads Programming Model

Xavier Martorell, Jesus Labarta, Nacho Navarro, Eduard Ayguade
Departament d'Arquitectura de Computadors (DAC)
Universitat Politecnica de Catalunya (UPC)
Gran Capita s/n, Campus Nord, Modul D6, 08071, Barcelona, Spain
{xavim, jesus, nacho, eduard}@ac.upc.es

Abstract

In this work we present the analysis, on a dynamic processor allocation environment, of four scheduling algorithms running on top of the nano-threads programming model. Three of them are well-known: uniform-sized chunking, guided self-scheduling and trapezoid self-scheduling. The fourth is our proposal: adaptable size chunking.

In that environment, applications are automatically decomposed into tasks by a parallelizing compiler which uses the Hierarchical Task Graph to represent the source application. The parallel code is an executable representation of this graph with the support of a user-level library (the nano-threads library).

The execution environment includes a user-level process (CPU manager) which controls the allocation of processors to applications. The analysis of the scheduling algorithms shows it is possible to provide enough information to the library to allow a fast adaptation to dynamic changes in the processors allocated to the application.

1. Introduction

Parallelization of sequential code has taken higher relevance since multiprocessors have been introduced in the industrial market. Applications can be rewritten in order to make them parallel or can be automatically restructured by a parallelizing compiler. Extensive work is now being developed in this area. [3][4][5] have proposed new languages or extensions (mainly based on C and C++) to help in the parallelization of applications by hand. They are based on asynchronous parallel functions that let the caller thread to continue while the function is executed by another thread of control. Programmers have to learn new syntactic constructs and new semantics to use these language extensions.

On the other hand, [16][13] propose to automatically parallelize applications written in standard languages (e.g.,

This work has been supported by the European Community under the ESPRIT project E21907 (NANOS) and the Ministry of Education of Spain (CICYT) under contracts TIC95-0492 and TIC94-0439.

C and FORTRAN). They are based on the nano-threads programming model, as defined in [14]. Nano-threaded applications are able to exploit both loop and functional parallelism. Using a parallelizing compiler such as Paraphrase-2 [15], the application is decomposed into parallel tasks from the largest to the smallest granularity. The Hierarchical Task Graph (HTG) is the internal representation used to record all the information available about control and data dependencies. The HTG makes possible to equally handle parallel loop iterations and parallel functions.

We have designed and implemented a prototype of the complete nano-threads execution environment based on a user-level library (the nano-threads library) and a user-level CPU manager that controls the allocation of processors to the running applications [9][12].

In this paper we center our attention in the study of the performance obtained by several dynamic loop scheduling algorithms. Three of them are well-known: uniform-sized chunking, guided self-scheduling and trapezoid self-scheduling. We have selected them from the scheduling algorithms studied in [10] to analyze their behaviour when ported to a new execution environment. Besides them, the fourth algorithm is our proposal called adaptable size chunking.

The rest of the paper is organized as follows: Section 2 presents the definition and basis of the nano-threads programming model. Design decisions, implementation of the user-level library, code generation and scheduling algorithms are shown in Section 3. Section 4 presents the execution environment and analyses the behaviour of the different scheduling algorithms. Finally, Section 5 concludes and presents future work.

2. The Nano-Threads Programming Model

In the environment defined by the nano-threads programming model [16] applications are automatically decomposed by a parallelizing compiler. The compiler identifies the maximum parallelism of the application through data and control dependence analysis and gener-

ates an intermediate representation of the parallel application in the form of a Hierarchical Task Graph (HTG). The compiler generates executable code from the HTG intermediate representation. This code is able to adapt to variations in the number of processors assigned to the application.

2.1. The Hierarchical Task Graph (HTG)

The HTG is an intermediate program representation built from a source program after control and data dependence analysis (see an example in Figure 1). The HTG structures the parallelism in levels of granularity. Each node in the HTG corresponds to a set of operations to be executed. Given a node in some level of granularity, it is possible for it to contain more nodes of the next (finer) level of granularity.

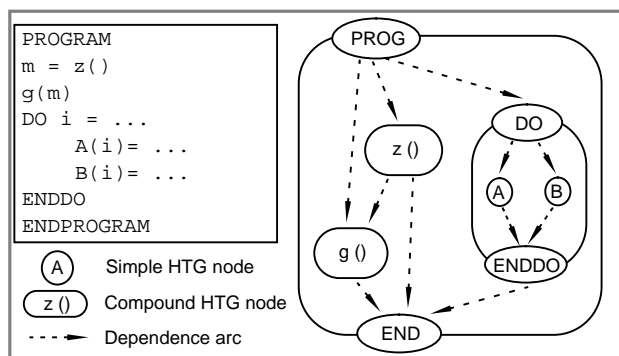


Figure 1: An example program and the associated HTG.

There are two types of nodes in the HTG: simple nodes and compound nodes. Simple nodes contain sets of operations that need to be executed sequentially. They represent the finest level of granularity and can not be further decomposed. The compiler will select the finest level of granularity to be generated having in mind the underlying architecture and the efficiency of the nano-threads package. Compound nodes contain complex computations like loops, conditional statements, etc.

Nodes at the same level of the hierarchy in the HTG are connected by arcs if there are data or control dependencies between any operation included in them. The connection of two nodes implies that they must be executed in sequential order.

Every compound node has an entry point. This is an input control subnode called the *start node*. All input dependencies for a node are directed to its start node. Internal nodes contained in a compound node are also subject to control and data dependencies. In every compound node there exists an end subnode that controls the output dependencies and it is the origin of them. That node is called the *stop node*.

2.2. Code generation from the HTG

A parallel program is obtained from the HTG representation of a parallel application. In our environment, the parallel code has the following characteristics:

- A C function is generated for both simple and compound nodes.
- Local variables are allocated on the stack.
- Output dependencies are embedded in the generated code as calls to the run-time library.
- Input data dependencies are represented by a per node counter of pending data dependencies, as in [2].

Following the previous scheme, the emitted code is an executable representation of the HTG. All services needed by the generated code such as creation of parallelism, control of dependencies, thread management, etc. can be provided by a run-time user-level thread package or by direct code injected by the compiler. In this work, we are using the first option.

2.3. The HTG execution mechanism

The execution of the parallel program consists in the execution, in some order, of the functions associated to the HTG nodes. The execution order ensures that all the dependencies for a function are satisfied before that function is executed. A nano-thread corresponds to an instantiation of an HTG node in the form of an independent user-level control flow.

The application maintains a user-level queue of ready to run nano-threads. The execution of an HTG begins with the main node being prepared for execution and inserted in the ready queue as a nano-thread. Virtual processors assigned to the application use auto-scheduling [13] to execute the nano-threads selected from the ready queue.

2.4. Application adaptability to the available resources

The overall execution of a nano-threaded application is able to adapt to changes in the number of processors assigned to it. The adaptation is dynamic, at run-time, and includes two important aspects: first, the amount of parallelism that a compound node generates is limited somehow by both the number of processors assigned to the application and the current amount of work already pending to be executed. And second, the application should be able to receive and free processors in conjunction with the operating system allocation decisions.

The nano-thread executing a compound node takes the decision whether to proceed in parallel or to execute itself (sequentially) the computations of the internal nodes, reducing the overhead of creation and management of such parallelism.

The operating system moves processors from one application to another following some policy (e.g., a priority-

based one [14]). Receiving new processors is not a problem for a nano-threaded application because they simply are going to pick up work from the ready queue and execute it.

The problem appears when the operating system decides to request processors from one application. The work the moving processors are performing in the source application is preempted and the termination of the HTG node will be delayed. To avoid preemption of useful work [7], we have designed a simple and efficient protocol between the operating system and the applications. The operating system asks for processors to the application giving a grace time to it to answer. When the application reaches a nano-thread scheduling point (a safe point, where the application is going to select the next nano-thread to be run), it tests for the request, releasing the processors, if necessary. The fine-grained decomposition of the application in tasks helps to detect in time the request of processors maintaining the overhead low [12].

If the application fails to respond in the given grace time, the operating system steals the requested processors from it. Even in this case, it is possible for the application to re-adapt to the imposed system conditions. Solutions to such problem will be presented in Section 4.

3. Design decisions and implementation guidelines

The goals in the library implementation have been to introduce the minimum number of services with a very simple functionality to minimize the package overhead and to maintain the implementation as close as possible to the outline established in the section 2. A complete description of the nano-threads library can be found in [11].

3.1. Nano-thread management

Nano-threads are represented by a fixed-sized structure which contains all the nano-thread information including its stack. As previously commented, the user-level scheduling policies help in maintaining the number of structures allocated below a limit.

Nano-threads are identified by a pointer that references the nano-thread descriptor. The interface to create a new nano-thread is:

```
struct nth_desc * nth_create (
    void (* routine) (),
    int ndep,
    struct nth_desc * successor,
    int narg,
    ... /* nano-thread arguments */);
```

which allocates a nano-thread structure and initializes its descriptor and stack. The *ndep* argument indicates how many data dependencies remain already unresolved for this nano-thread. The nano-thread will start its execution at the supplied *routine*. The *successor* argument is filled with another nano-thread that is data-dependent on this one. It

will be automatically enqueued when this terminates. The nano-thread stack is setup with the *nano-thread arguments* to simulate a call to the *routine*.

Other library functions help in the management of the nano-threads when they have already been created:

```
void nth_to_rq (struct nth_desc *);
enqueues the given nano-thread in the ready queue.

void nth_depadd (struct nth_desc *, int);
atomically adds new dependencies to the nano-thread.
```

3.2. Code generation for loops: nano-thread bursts

If the compiler determines that the iterations of a loop are independent, its parallelization consists on the generation of two functions:

- The body function: it contains the code generated for the loop body. Its general interface is

```
void body_func (
    int lower_limit,
    int upper_limit,
    ... /* other arguments */);
```

It receives the lower and upper limits of iterations to be executed. This interface facilitates to split the loop iteration space into chunks.

- The loop control function: it is in charge of the evaluation of the loop condition and invokes the body function as many times as necessary. In addition, it decides whether to execute it in parallel or not. The interface of this function depends on the compound HTG node where it is enclosed.

When the number of iterations in a loop is large enough and the body has considerable cost nano-threads are created in *burst mode*. All nano-threads created within a burst are assigned work from the loop, except one. This one is the dispatcher nano-thread and it is inserted in the ready queue as a scheduler thread. When it executes, it decides if it is possible to create another burst or it is better to proceed with a sequential execution. In this way, we limit the overhead of the nano-thread creations in terms of memory usage because we do not create all nano-threads in only one burst. This work generation scheme differentiates our work from previous ones (like Filaments [5] and Cilk [3]).

Three calls have been designed to give support to such loop scheduling. The interface is as follows:

```
struct nth_desc * nth_burst_create (
    int ndep);
void nth_burst_wait (
    struct nth_desc * nth_burst);
void nth_block ();
```

Figure 2 shows an example of the control and body functions generated for a parallel loop. First, the code creates the nano-thread associated to the stop node of the loop node (line 01 in Figure 2). If there are enough resources to proceed in parallel (line 03), a first burst of nano-threads is created (lines 04-08). On each burst *CHUNKS_PER_BURST* nano-threads are created. Each

nano-thread is given *ITER_PER_CHUNK* iterations and inserted in the ready queue using the *nth_to_rq* primitive. The successor of each nano-thread is set to be the stop node of the loop. For this reason *nth_adddep* is used to increment the number of pending dependencies for the stop node. Finally, the dispatcher nano-thread is created using the *nth_block* function (line 10). This function lets a nano-thread to wait for the termination of the parallelism that it has created.

```

...
01. nth_end_loop = nth_burst_create ();
02. for (i=0; i<N; ) {
03.   if (IN_PARALLEL (CHUNKS_PER_BURST)) {
04.     nth_depadd (nth_end_loop,
05.                CHUNKS_PER_BURST);
06.     for (k=0; k<CHUNKS_PER_BURST;
07.          k++, i+=ITER_PER_CHUNK) {
08.       nth = nth_create (chunk_func, 0,
09.                        nth_end_loop, nargs, i,
10.                        i+ITER_PER_CHUNK , ...);
11.       nth_to_rq (nth);
12.     }
13.   }
14.   else {
15.     chunk_func (i, i+ITER_PER_CHUNK, ...);
16.     i += ITER_PER_CHUNK;
17.   }
18. }
19. nth_burst_wait (nth_end_loop);
...

20. void chunk_func (int lower, int upper, ...)
21. {
22.   int i;
23.   for (i=lower; i<upper; i++)
24.     loop_body;
25. }
26. }

```

Figure 2: Code generated for a parallel loop.

In case the test for parallel execution fails, the loop proceeds sequentially for one chunk (lines 13-14). The loop is executed till all iterations are exhausted and its stop node is invoked through the *nth_burst_wait* call (line 17). The function in lines 20-26 contains the body of the loop.

Observe that the dispatcher nano-thread is created after all other threads in the burst are created. An interesting variation on this consists in creating the dispatcher near the middle of the burst in order to begin the generation of the next burst before the previous one has exhausted and thus get better performance avoiding the ready queue to become empty. This functionality can be achieved splitting the *nth_block* call into two functions. Also, a very simple improvement consists in the substitution of the sequential version by a more limited generation of parallelism.

3.3. Scheduling algorithms

We have tested the behaviour of several scheduling algorithms [10]. We have adapted the algorithms to our environment in order to avoid the generation of all the work first. Instead, we generate bursts of work following the burst scheme presented in the previous section, also

limiting the amount of memory used by the nano-thread structures.

The first algorithm is the uniform-sized chunking (Uni). We use it as the reference to which compare any other scheduling algorithm implemented in our environment. It is the algorithm that introduces the highest overhead, but it also is the algorithm that provides the best adaptability of the application to a varying number of processors because of its finer granularity.

Uni generates a burst of chunks large enough to fill all the processors assigned to the application and then waits for the burst to exhaust. When the burst is nearly exhausted, another burst is generated. Every chunk in the burst comprises the same number of iterations, usually small enough to properly adapt to the number of processors, but large enough to maintain the overhead below 10% of the sequential application time. Figure 3a shows an example of the work generation scheme for this algorithm using three processors. In the example, the minimum number of iterations to maintain the overhead lower than 10% is 2, which we call the *base* size of the chunk.

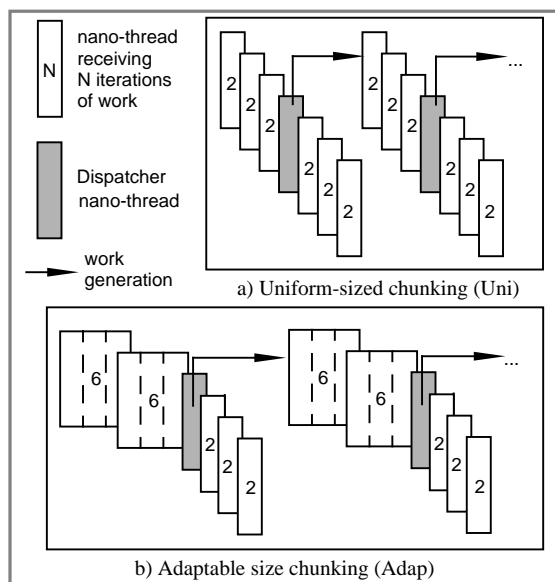


Figure 3: Work generation using burst-based scheduling algorithms.

The second algorithm is the guided self-scheduling algorithm (GSS). GSS begins assigning large chunks. Each processor receives as much work as the number of remaining iterations divided by the number of processors (N/P). Chunks quickly decrease in size till all the work has been generated. GSS is the algorithm that introduces less overhead of those presented here.

The third algorithm is the trapezoid self-scheduling (Trap). It begins assigning chunks smaller than the GSS algorithm ($N/4P$). Chunks decrease also slower than in GSS (at a rate of $N/(32*P*P)$).

The last algorithm (Figure 3b) is the adaptable size chunking (Adap). It generates work using two chunk sizes,

one for each part of the burst. First, it generates $P-1$ large chunks. These chunks ensure that all processors except one get a large amount of work with low overhead. The second part of the burst consists of a collection of P small chunks that contains as much work as one large chunk. In the example 3b, the base size (the small chunk size) is 2 and the large size is $6 (= 2 * P)$. In Section 4.4, we show how this technique lets the application to adapt to the system conditions with small overhead.

4. Analysis of the scheduling algorithms

We have implemented a user-level process (the CPU manager) that controls the processors assigned to the applications. Related work on this topic is [9]. The main goal is to be able to measure the impact of the scheduling algorithms presented in the previous section.

We have implemented and tested three versions of the protocol between the CPU manager and the applications. We are going to present them in the following subsections along with the experimental measures we have obtained.

4.1. Dynamic processor allocation execution environment

When the application starts, it requests some number of processors to the CPU manager. If the request is feasible, the CPU manager grants them. During the execution of the application the CPU manager asks for (or steals) and returns processors from/to the application following some determined plan. In the current implementation, it periodically requests and returns one processor (every $t_{ncpu-changes}$ time).

The CPU manager controls the time every processor is executing in the application. It requests processors to the application following the protocol explained in Section 2.4. This protocol is implemented through shared memory between the CPU manager and the applications. At every processor request, the CPU manager waits for the application to respond during a grace time. If the application releases the processor in time, the CPU manager gets it and allocates it to another application. When the application fails to respond in the given grace time, the CPU manager steals a processor from the application.

The main difference between a processor release (by the application) and a processor stealing (by the CPU manager) is that in the latter case some work in the application is preempted and the introduced delay may reduce the application performance [7][8]. In the first case, no work is delayed because the user-level library ensures that processors are released at nano-thread scheduling points, where they are not processing useful application work.

The core of the user-level library contains the nano-thread scheduling loop that searches for work in the ready queue. This loop is executed by as many kernel threads as

processors are assigned to the application as it is also done by [18]. Kernel threads are supplied by the operating system. In the current implementations, they are Mach kernel threads and IRIX sprocs. The goal is to maintain a one-to-one mapping between physical processors and kernel threads during the overall execution of the application.

In this way, releasing a processor (by the application) or stealing it (by the CPU manager) simply consists in suspending the kernel thread executing on it. The main difference is that the application does it at a nano-thread scheduling point, but instead, the CPU manager probably does it at an unsafe point, so work can be delayed. Returning a processor consists in resuming the previously suspended kernel thread.

We have implemented the nano-threads library on top of Mach 3.0 microkernel [1]. We use a four i486 multiprocessor architecture (DEC433MP) with 32 Mb. of main memory and 256 Kb. of coherent cache per processor. Our implementation of the nano-threads library is based on the Quick Threads package [6]. The library has been ported to the MIPS R10000 architecture running the IRIX O.S.

4.2. First results

Figure 4 shows the behaviour of the four scheduling algorithms under analysis running in the environment explained in Section 4.1. The figure presents the execution time of the parallel code of a Jacobi iteration used to solve a linear system of equations based on a matrix of 480 rows per 480 columns in the DEC433MP machine. The resolution of the linear system spends 250 iterations.

Labeled horizontal lines are the representation of the execution time of the sequential (no nano-threaded) version of the same application and the reference times taken from the parallel versions based on the Uni (highest overhead) and GSS (lowest overhead) algorithms using 1, 2 and 3 processors. Reference times for Trap and Adap algorithms fit between those of Uni and GSS.

The CPU manager requests and returns one processor every $t_{ncpuchanges}$ time. In the figure, 40, 400 and 4000 milliseconds are used. The grace time is set to 3 milliseconds. In our environment, applications are able to adapt in several milliseconds, while other implementations have been more oriented to adapt within larger periods of time [18].

Different scheduling algorithms perform different in such environment. The analysis follows:

- Uni adapts well to the processor variations because the small size of the chunks makes it to generate a large number of nano-threads and, thus, there is also a large number of nano-thread scheduling points to test for processor requests. The frequency of changes does not affect this algorithm, except for the decreasing overhead introduced by the operating system when there are less processor movements (at 4000 ms.)

- On the other hand, both GSS and Trap generate large chunks at the beginning of work generation. The large chunks do not always give the library the chance to answer in the given grace time. In this case, the processor is stolen and application work is preempted. Both GSS and Trap perform worse when the time the processor is stolen is larger because the introduced delay is also larger. Trap scheduling usually performs better than GSS because Trap generates chunks smaller than GSS.
- Finally, the Adap algorithm performs like Uni, but the overhead is smaller because of the larger chunks generated.

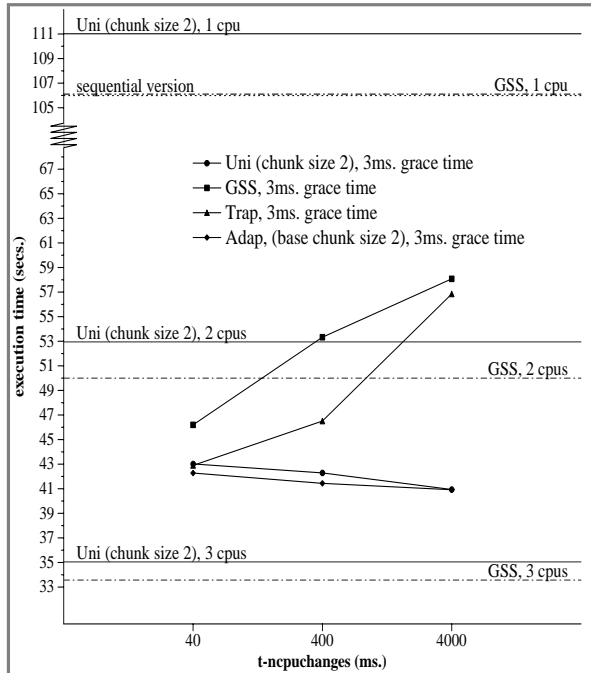


Figure 4: Effect of the processor assignment in the execution time (dynamic processor allocation execution environment).

4.3. O.S. scheduling vs. stealing notifications

We have tested two attempts to solve the work preemption problem. First, having in mind that in a standard operating system the preemption of processors from the applications is not notified, we have implemented a version of the CPU manager that simply steals physical processors without giving any information to the applications. This means that applications work with more virtual than physical processors. We call this first attempt the O.S. scheduling environment.

The second attempt (and the actual solution to the problem) consists in the notification of which processor has been stolen to the application after a grace time expiration. The nano-threads library detects the notification through one of the remaining processors, when it finally reaches a nano-thread scheduling point. Then, it is able to answer to the CPU manager, suspending the execution of the current

virtual processor and resuming the stolen one. The executing physical processor switches immediately to continue the execution of the previously preempted work. We call this second attempt the stealing notifications environment.

4.4. Performance evaluation and analysis

Figure 5 shows the results of both the O.S. scheduling and the stealing notification environments. Sequential times are the same as those presented in Figure 4. Dotted lines represent the behaviour of the Jacobi iteration when using the O.S. scheduling environment. Performance decreases using any of the studied algorithms because the operating system applies its scheduling policies between physical and virtual processors when the application is running with more virtual than physical processors. This situation is also avoided in other works, like in [18]. It is already known that the operating system general-purpose scheduling policies usually get bad performance when applied to parallel applications.

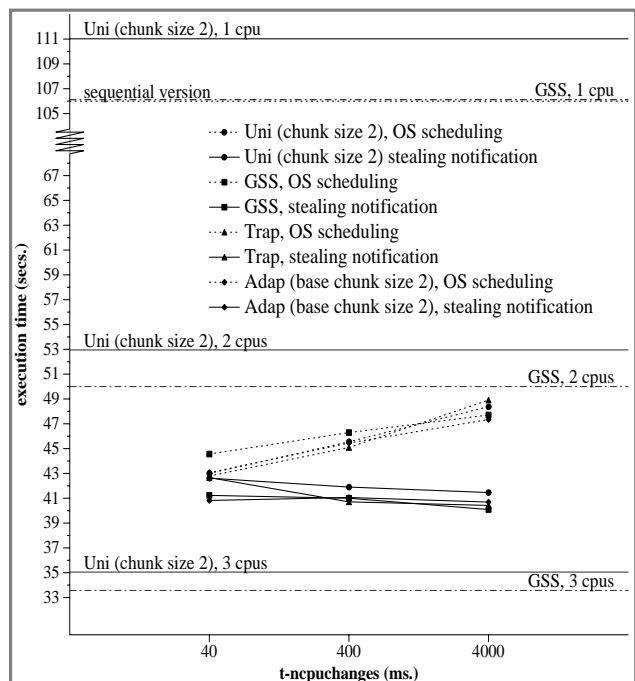


Figure 5: Effect of the processor assignment in the execution time (O.S. scheduling vs. stealing notifications).

Solid lines in Figure 5 represent the Jacobi iteration performing in the stealing notification environment. Compare the results with those in Figure 4. Uni and Adap algorithms maintain the same performance that in the simple environment showed in Figure 4 because these algorithms never have to re-adapt using the stealing notification. Instead, GSS and Trap algorithms now perform as well as if they would correctly adapt to the processor variation. Observe that although every time that GSS or Trap fails to respond in the given grace time they have to cause a context switch

to resume the preempted virtual processor, this is better than to let the operating system schedule the virtual processors on top of the physical processors, causing as many context switches as the operating system scheduling policies dictate.

5. Conclusions and future work

We have analyzed four scheduling algorithms running on top of the nano-threads execution environment. The environment consists in a user-level library to support parallelization and a CPU manager which is able to periodically request and return processors from/to the applications. Results show that nano-threaded applications are able to adapt to variations in the number of assigned processors. The fine-grained decomposition of the application through the Hierarchical Task Graph helps the detection of the processor requests in the given grace time.

Although the protocol used between the CPU manager and the application is important, the scheduling algorithms used to generate parallel work are also relevant in the efficiency of the applications. In a simple dynamic processor allocation environment, scheduling algorithms that generate large chunks of work (i.e. GSS and Trap) prevent the correct adaptation to a processor reassignment. We also show that using stealing notifications, these algorithms adapt as well as Uni and Adap. In any case, it is better to control processors from the application (at user level) than from the operating system, because the operating system general-purpose scheduling policies do not satisfy the parallel applications needs.

More experiments, measurements, behavioural studies and applications running on top of the nano-threads programming model are needed. We plan to trace the execution of the parallel applications based on the nano-threads library to identify the major drawbacks and bottlenecks of the current implementation. The porting to the SGI R10000 architecture has to be completed with the port of the CPU manager and we also plan to port the execution environment on top of the Chorus microkernel [17].

6. References

- [1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian and M. Young, "Mach: A New Kernel Foundation for UNIX Development", Proc. of the Summer 1986 USENIX Conference, July 1986.
- [2] C. J. Beckmann, "Hardware and Software for Functional and Fine Grain Parallelism", Ph.D. thesis, Dep. of Elec. and Comp. Eng., Univ. of Illinois at Urbana-Champaign, 1993.
- [3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall and Y. Zhou, "Cilk: An Efficient Multi-threaded Runtime System", Proc. of the Fifth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP '95), Santa Barbara, California, July 19-21, 1995.
- [4] R. Chandra, A. Gupta and J. L. Hennessy, "Data Locality and Load Balancing in COOL", Fourth ACM SIGPLAN Symp. on the Principles and Practice of Parallel Programming (PPoPP), pp. 249-259, May 1993.
- [5] D. R. Engler, G. R. Andrews and D. K. Lowenthal, "Filaments: Efficient Support for Fine-Grain Parallelism", TR 93-13a, Dept. of CS, University of Arizona, Tucson, 1993.
- [6] D. Keppel, "Tools and Techniques for Building Fast Portable Threads Packages", Technical Report UWCSE 93-05-06, University of Washington, 1993.
- [7] A. Gupta, A. Tucker and S. Urushibara, "The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications", Proc. of 1991 ACM SIGMETRICS Conference, May 1991.
- [8] S. T. Leutenegger, M. K. Vernon, "The Performance of Multiprogrammed Multiprocessor Scheduling Policies", Proc. of ACM SIGMETRICS Conf., Vol. 18, No. 1, May 1990.
- [9] C. McCann, R. Vaswani and J. Zahorjan, "A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors", ACM Transactions on Computer Systems, Vol. 11, No. 2, pp. 146-178, May 1993.
- [10] E. P. Markatos, T. J. LeBlanc, "Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors", Proceedings of the Supercomputing-92, 1992, pp. 104-113.
- [11] X. Martorell, J. Labarta, N. Navarro, E. Ayguadé, "Nano-Threads Library Design, Implementation and Evaluation", Dept. d'Arquitectura de Computadors - Univ. Politècnica de Catalunya. Tech. Report: UPC-DAC-1995-33, Sept. 1995.
- [12] X. Martorell, J. Labarta, N. Navarro, E. Ayguadé, "A Library Implementation of the Nano-Threads Programming Model", Proc. of the Second International Euro-Par Conference, vol. 2, pp. 644-649, Lyon, France. August 1996.
- [13] J. E. Moreira, "On the Implementation and Effectiveness of Autoscheduling for Shared-Memory Multiprocessors", Ph.D. thesis, Department of Electrical and Computer Engineering, Univ. of Illinois at Urbana-Champaign, 1995.
- [14] C. D. Polychronopoulos, "Multiprocessing versus Multiprogramming", Proceedings of the International Conference on Parallel Processing, St. Charles IL, August 1989.
- [15] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, Chia Ling Lee, B. Leung and D. Schouten, "Parafraze-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors", International Journal of High Speed Computing, Vo. 1, No. 1, 1989.
- [16] C. D. Polychronopoulos, N. Bitar, S. Kleiman, "nano-Threads: A User-Level Threads Architecture", CSRD Technical Report, 1993.
- [17] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Léonard and W. Neuhauser, "Overview of the Chorus Distributed Operating System", Proc. of USENIX Workshop on Micro-kernels and Other Kernel Architectures, abril 1992.
- [18] A. Tucker, A. Gupta, "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors", ACM OS Review, Vol 23, Num 5, Dec. 1989.