Global optimisation and Scheduling for Asynchronous Processor Architectures

2nd. Year Report

Salvador Sotelo Salazar

December 2, 1997

Abstract

This work addresses the problem of compiler optimisations for ILP in asynchronous architectures. Instruction scheduling for a Micronet-based asynchronous target presents interesting challenges. First, the model of the underlying target is more complex: the functional units have different latencies which range over different intervals. Secondly, the dynamic behaviour of the asynchronous architecture makes it impossible to consistently predict the optimal order of execution.

This second year report presents the work done over the last year concerning the local scheduler, the current work towards a global scheduler, and an early outline of my thesis.

1 Introduction

Asynchronous architectures differ from their synchronous counterpart in that the central control unit in the processor does not have a central clock to synchronise the operations of the functional units. The method that asynchronous architectures use to control the flow of instructions and data along the datapath is a handshake. The stages in an asynchronous pipeline take different times depending on the latencies of the individual stages.

The datapath of MAP (Micronet-based Asynchronous Processors) architectures [1][2] is modelled as a network of functional units that communicate in an asynchronous manner. This model exhibits fine grain concurrency, both spatial and temporal, so that each instruction takes the necessary stages for its execution and only for the time needed. In the synchronous world, the instructions have to go through all the stages and even if they complete one stage sooner they have to wait for the next cycle to start another operation, thus keeping that resource busy but not useful.

It has been shown that reordering the instructions of a program in a RISC processor is quite effective in achieving better performances [11][15]. The idea of arranging the order of the instructions in a code is to exploit the benefits of the pipeline and avoid stalls. This is achieved by the fact that the compiler knows at any time the state of the datapath and the *latencies* of the instructions are fixed in terms of multiples of cycles. This determinism in synchronous RISC processors has helped substantially the scheduling theory for improving execution times.

However, in the asynchronous approach this determinism is not valid. The latencies in such architectures do not have a precise value and vary over a range. Therefore the way to schedule a program given this model is a new field for research. An example can be found in [3] using a list scheduler. We have looked at the problem in a novel way without using a list scheduler.

This second year report presents the work conducted in the static analysis of programs. Firstly, we describe the work done on the local scheduler: the heuristic on which it is based, description of the algorithm, characteristics of the scheduler and some performance comparisons. Secondly, we describe the work done so far on the global scheduler and we point out the work to be done on a global optimiser. Finally, we propose an outline for the thesis.

2 Local scheduler

2.1 Related work

Scheduling code on synchronous RISC processors is simplified by the fact that the latencies of pipeline stages are often considered equal and the instruction's execution times tend to be one cycle. For example, in [11] the proposed scheduler has an $\theta(n^2)$ complexity which considers only 0 or 1 cycle latencies (where n represents the number of instructions in the basic block). Palem and Simons

[24] present a polynomial order scheduler based on the *rank* algorithm that does not guarantee a feasible schedule if latencies are greater than one. Proebsting and Fisher [25] propose a linear time optimal code scheduler for delayed-load architectures with all instruction latencies of one cycle duration.

The balanced scheduler [15] introduces an algorithm to measure *load* level parallelism that produces beneficial results when the load latency is unknown, with a worst case complexity order of $\theta(n^2\,\alpha(n))$ [1]. This is one good attempt to look at the non-uniformity of the architectures. However, it has been shown in [17] that the balanced scheduler needs different and important considerations for an asynchronous target, such as considering if one instruction's unit has been used previously.

Taking into consideration that the order of the scheduling problem is NP-complete [31] and the non-determinism of MAP asynchrony makes it hard to predict its behaviour, the complexity of a scheduler algorithm could be expected to be high using traditional synchronous approaches. We next show the concept behind the *Penalise True Dependencies* (PTD) scheduler and its relevance to asynchronous architectures.

2.2 PTD scheduler

2.2.1 The Concept

The concept of penalising instructions was first introduced in [28] and basically "penalises" two instructions when the second one needs the result of the first one in a true data dependency. Depending on the type of instructions, the value of the penalty may vary. For example, the cost of a true dependency from a memory load instruction is higher than one between registers. As our work has focused on architectures where the units tend to have different values over a range in their latencies, the way to penalise true dependencies is done according to the description of the architecture, i.e. the number and type of units and their range in latencies.

The result of applying penalties with different weights to instructions gives us a scalar measure relative to the way the code has been scheduled. In [4], the penalty idea has been extended to resource dependencies when instructions are consecutive, and for the true data dependencies, for instructions that are not strictly consecutive. For example, for the former, we apply a penalty to a pair of instructions of the same type if there are not sufficient units of that particular type.

The paper also shows results on how the criteria of penalising instructions is sufficient to be considered the main heuristic of the local scheduler. The penalty (PTD) measure was directly proportional to the makespans forming an interesting and useful pattern.

[1] α is the inverse of the Ackerman function.

2.2.2 The algorithm

The main idea of the PTD scheduler is to minimise the number of penalties on each basic block. This is done by inserting independent instructions between those who share a penalty. Furthermore, when an instruction is moved, it has to be checked that by doing so it does not introduce another penalty to the instructions around it, and to the instructions where it is going to be moved. This is to guarantee that after every transformation the penalty measure is always reduced.

The scheduler computes first the penalty measure, which is the addition of all the penalties, while marking the instructions. As there is no way to know beforehand which is the minimum number of penalties -knowing it would mean to know an optimal schedule- the algorithm must try to reduce any penalty. If after one pass through the code there is at least a reduction of one unit in the penalty measure, the algorithm continues for further reductions. If the penalty measure stays constant after a pass, we have to assume that there are no possible reductions left and therefore the algorithm has to terminate. This decision is relevant and unique for the sake of the termination of the algorithm.

The algorithm has been partitioned in two main phases: the first one deals only with resources penalties and the second with true data dependencies. As the scheduler is sensible to the way the instructions are ordered, the order in which these phases are selected is quite important. Given the number and high cost of true data dependencies in a code, the penalties resulting from them have a higher impact in the overall time than resource penalties. Thus, we have opted for the first phase the reduction of resource dependencies and for the second, the reduction of true data dependencies. Figure 1 shows the first and the second phase of the scheduler.

```
void PTD_first_phase(dagnodes *root) {
  int measure, last_measure;

  measure = PTD_measure(root, first_phase);
  if (measure > 0)
    do {
      node = root;
      last_measure = measure;

      while (node != NULL) {
        if (node -> PTD.penalised > 0)
          PTD_arrange_left_unit (node);
        if (node -> PTD.penalised > 0)
          PTD_arrange_right_unit(node);
        node = node -> next;
      }
      measure = PTD_measure(root, first_phase);
    } while (measure < last_measure);
}
```

```
void PTD_second_phase(dagnodes *root) {
  int measure, last_measure;

  measure = PTD_measure(root, second_phase);
  if (measure > 0)
    do {
      node = root;
      last_measure = measure;

      while (node != NULL) {
        if (node -> PTD.penalised > 0)
          PTD_arrange_left_data (node);
        if (node -> PTD.penalised > 0)
          PTD_arrange_right_data(node);
        node = node -> next;
      }
      measure = PTD_measure(root, second_phase);
    } while (measure < last_measure);
}
```

Figure 1: First and second phase routines of the PTD scheduler.

The main difference in the functions of Figure 1 is that they call different routines for the transformations: *arrange_unit* or *arrange_data*. These functions are in charge of performing the movement of instructions while respecting all data dependencies. In the case of *arrange_units* routines, the only consideration is to move instructions to reduce resource penalties, while for the *arrange_data* routines, their consideration is to reduce penalties from true data dependencies only.

We can see the schedule from a basic block as lines of instructions where the first instruction represents its *entry* and the last represents its *exit*, and the flow of the program runs from the entry to the exit. The scheduler can either move instructions from the left (towards the entry) or from the right (towards the exit), from a pair of instructions that share a penalty. Figure 2 shows one of the four functions that perform a transformation. *PTD_arrange_left_data* seeks the first independent instruction to the left of the penalty that reduces the penalty measure.

```
void PTD_arrange_left_data(dagnodes *nodedag) {

  dagnodes *aux1 = nodedag -> prev, *aux2;
  int ind_nodes = 1, cond_out = 0;

  if (aux1 != NULL)
    aux1 = aux1 -> prev;

  if (independ(nodedag -> prev, nodedag) && valid_swap_left(nodedag, aux1)) {
    update_node(nodedag);
    update_node(aux1);

    swap(nodedag -> prev, nodedag);
  }
  else {
    while (aux1 != NULL && !cond_out) {
      aux2 = aux1 -> next;
      while (aux2 != nodedag -> next && ind_nodes) {
        ind_nodes = ind_nodes && independ(aux1, aux2);
        aux2 = aux2 -> next;
      }

      if (ind_nodes && valid_move(nodedag, aux1))
        ind_nodes = 0;

      if (ind_nodes) {
        cond_out = 1;

        update_node(nodedag);
        update_node(aux1);

        move_ahead(aux1, nodedag);
      }
      ind_nodes = 1;
      aux1 = aux1 -> prev;
    }
  }
}
```

Figure 2: The *PTD_arrange_left_data* routine.

The four routines can call either function *swap* or *move_ahead* to reduce a penalty. The former one (line 10) performs a swap between the first penalised instruction and its previous one. Line 7 checks that both instructions are independent and that the transformation reduces the penalty measure. The latter function (line 25) moves an independent instruction pointed by *aux1* ahead of *nodedag* to reduce the penalty.

Routine *update_node* (lines 8-9 and 23-24) just updates the penalty measure and the penalties markings for the involved instructions, when a transformation is done.

2.2.3 Complexity

The complexity of the scheduler is literally $\theta(2\,nec + 2\,nc + 2\,n)$, where n is the number of instructions in the basic block, e is the number of penalties and c is a small constant ($c = 2, 3, 4$). The complexity time of computing the number of penalties is just the order of $\theta(n)$. This is reflected in the term $2\,nc$ and $2\,n$. The term $2\,nec$ reflects the actual complexity of the *PTD_arrange_left* and *PTD_arrange_right* functions in Figure 1.

The upper bound, which is represented by a pure sequential code, is $\theta(n^2)$, with $e = n - 1$ and $c = 2$. Conversely, the lower bound is represented by a pure independent code and is the order of $\theta(n)$, with $e = 0$.

Analysing the equation above, the complexity of the scheduler can be reduced to $\theta(ne)$. However, in general conditions, as the algorithm progresses the number of penalties is reduced and therefore n becomes bigger than e, which means that the complexity can be considered as $\theta(n)$.

2.2.4 Results

The PTD scheduler was compared against other two local schedulers, the *balanced* scheduler [15] and the original *list* scheduler from Gibbons and Muchnick (GM)[11]. The three schedulers were performed after register allocation and were tested over a set of benchmarks. An event-driven simulator for the MAP architecture was used to simulate the benchmarks [17]. The architecture's principal feature was that the latencies from the units had different ranges. This is to reflect an architecture with a high degree of uncertainty.

The simulations were done with an architecture with one memory unit, one logical unit, one branch unit and one, two, three and four arithmetic units (1 AU, 2 AU, 3 AU and 4 AU respectively). The MIPS-like code produced by the SUIF compiler clearly has a lot more of arithmetic instructions that motivated us to scale the number of these type of units to see the scalability of the algorithms.

The set of benchmarks chosen was the set of Livermore benchmarks [8] which are loop oriented (few basic blocks), and a set of programs with a larger number of basic blocks that we will refer as the control intensive set [2].

[2] In the near future we will modify the simulator in order to run traces of bigger programs like the Spec95 benchmarks.

The results from the simulations show that the PTD scheduler minimises the total stall time from the issue unit quite favourably when compared to other approaches, and in some cases better. This leads, in general, to better performances over the set of benchmarks. The complete set of results can be found in the Appendix A; each result represents the average of 5 simulation runs.

These results and the description of the scheduler itself are presented in [4]. This paper was presented at the 3rd. International Euro-Par Conference in Passau, Germany in August 1997.

2.2.5 Limitations

The main limitation of the local scheduler is that in few cases there are nested penalties. In order to eliminate them, the algorithm has to perform two or more transformations and not necessarily with the same instructions. This means that there is no independent instruction (in that basic block) that by moving it in between the penalised instructions, the penalty measure would be reduced. The scheduler then must be capable of identifying instructions that could be moved to a place where the penalty measure stays constant, and from there evaluate if there is a second instruction that could be placed in the desired position to reduce the penalty. If this combination of instructions fails to satisfy the conditions, the transformations must be reversed and the process of testing other combinations has to continue. The major problem though, is that we do not know how many transformations have to be done for a particular penalty, and if we add that the cost of doing a two-instruction transformation has an upper bound of $\theta(n^2)$, the task simply would not be worth[3].

Another point to mention is that the PTD scheduler is non-deterministic. In other words, it is sensible to order of input code and therefore produce different, but equivalent outputs. Although higher penalties are sorted first than lower ones, the algorithm must choose among the penalties that share values. This decision leads to different "paths" during the transformations, giving at the end different schedules. We believe that this drawback is compensated by the complexity and the results achieved.

3 Proposed work

It is well understood that the available ILP within basic blocks is insufficient to maintain a high level of resource utilisation[5][6][10][14][19]. To achieve better performances than local optimisations we need to look beyond the basic block boundaries in order to perform global optimisations. The back-end of the compiler is composed of two independent layers of optimisations as Figure 3 depicts it.

Section 3.1 describes the work and issues around the global scheduler, while section 3.2 describes the future work on the global optimiser and its concerning issues.

[3]For a three-instruction transformation the upper bound would be $\theta(n^3)$.

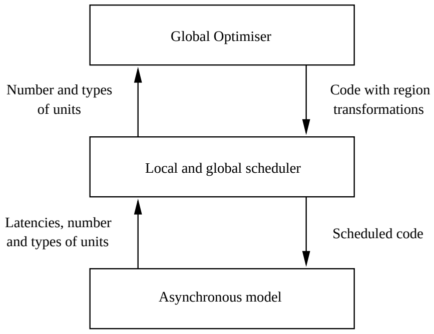

Figure 3: Different levels in the optimising framework

3.1 Global scheduler

We have seen throughout last section that the success of the PTD scheduler stems from the fact of reducing all the penalties. The need of a global scheduler capable of moving instructions between different basic blocks to reduce the remaining penalties is therefore apparent.

3.1.1 Related work

The work done in the field covers a wide area in optimisation techniques over different levels of program representations. One can distinguish program optimisations within two main areas: optimisations which are performed independent of the nature of the architecture (synchronous/asynchronous or single/multiple processor), and optimisations which are performed with some notion or a model of the target. We can see the former approach as an *optimiser* (see section 3.2), while the latter being the *scheduler* (global and local for our purposes), as it can be seen in Figure 3.

For example, the term *code motion* that is used in compiler optimisations can be applied both in the optimiser (expressions or intermediate code) and the scheduler (assembly code). At a high level, the main idea is to eliminate redundancies in the code: in [16], the movement of expressions across basic blocks is performed to remove assignment and expression redundancies; in [7], a linear-time implementation of global code motion outperforms the use of conditional constant propagation and partial redundancy elimination. It is clear that these optimisations improve the quality of the code quite independantly of the type of architecture.

At a lower level, the movement of instructions is used to increase the parallelism or ILP in the code. In this case, the heuristics and conditions to allow such transformations differ in practice with the target. Bernstein and Rodeh [6], for example, use a control and data dependence representation (Program Dependence Graph) to apply global code motion that works in innermost loops. Another approach has been proposed by Mahadevan and Ramakrishnan, where with the use of profile information, instructions can freely move across basic blocks within a region respecting *legality*, *safety* and *desirability* conditions [21].

3.1.2 Location of our work

Traditionally, compilers consider the function as the unit of compilation. The reason for this is that functions are a self contained entity and therefore provide a convenient way for partitioning the compilation process. The process of compilation may include optimisations, prepass scheduling, register allocation and postpass scheduling. Although there have been attempts to take other compilation units, like in [13], we will first concentrate our work with function-based compilation.

Each function usually contains several basic blocks which are connected with each other to form *regions*. The term 'region' has different interpretations in the literature, but for us, we mean a group of blocks that are strongly connected.

3.1.3 Definitions

A *region* is a part of a program in which these basic blocks are strongly connected. Let A and B be two basic blocks of a region. A is said to "dominate" B (A *dom* B), if for all the paths from the region's entry to B, A appears in them. Similarly, B "post-dominates" A (B *post* A), if B appears on all the paths from A to the region's exit. Figure 4 (a) depicts a region with its entry and exit nodes added. We can see that block 1 dominates all the other blocks because from the entry we find 1 in every path to any particular block. Similarly, block 7 post-dominates the rest of the other blocks as it appears from any block to the region's exit.

If both conditions hold true for blocks A and B, it is said that A is *equivalent* to B [6]. This is an important characteristic because it means that should A be executed, then B will definitely be executed. In practice, A and B can be executed even in parallel, as long as they respect their data dependencies. They are said to be completely *control-independent*. In the example, only blocks 1 and 7 fullfill both definitions, thus they are equivalent. The dotted line in Figure 4 (b) shows graphically the equivalence property between blocks 1 and 7.

3.1.4 Local scheduler extension

The natural extension for the local scheduler is to design and implement a global scheduler based on the PTD heuristic. We have shown that by reducing the number of penalties within basic blocks we get good performance improvements.

The dominator and post-dominator definitions allow us to perform code motion beyond basic blocks without *speculation* and without *duplication*. Speculation here refers to the movement of an instruction above a branch or below a join; when an instruction is moved below a branch or above a join, there is a need to duplicate it in order to preserve the semantics of the program. We will pay attention to the impact of speculation and duplication in the processor performance: too much

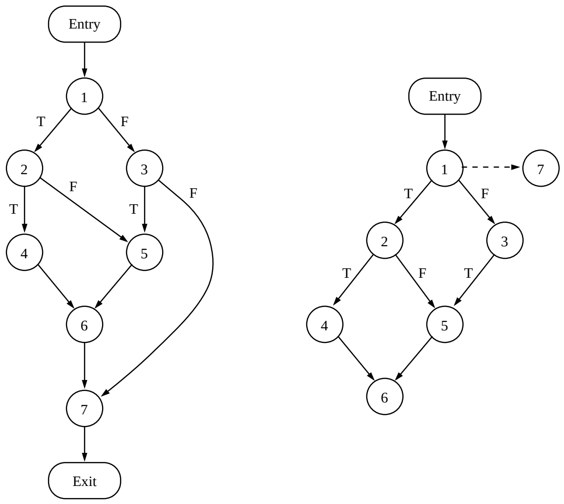

Figure 4: (a) CFG of a region and (b) its reduced representation

speculation leads to code with many instructions being executed needlessly, while duplication leads to code expansion. We will need the use of profile information to guide us with the transformations.

The local PTD scheduler minimises each basic block penalty measure by performing local movement of instructions. Once this has been done, the global motion of instructions can be applied in the same way to reduce penalties from two different blocks. As each basic block has its own measure, we can see a *global measure* as the total amount of penalties in a region. Since the idea is to reduce each basic block's penalty measure, it would appear that getting the minimum from each one would represent the minimal global measure. The possible advantage of this is that we are not required to apply the local scheduler after the global transformations, although we will compare both effects.

These ideas differ from [6] in several ways. First, their global scheduler is basically a *global list scheduler* where the ready list consists of instructions from a basic block and its equivalent blocks. Secondly, they apply two heuristics to each instruction one called *delay heuristic* that gets the maximum accumulative delay from its successors, and a *critical-path heuristic* that computes a measure of time needed to complete the execution of the successors. A list of priorities is given in the paper to select between ready instructions at the same time in the scheduling process. Lastly, they use the local scheduler at the end to correct the global decisions over the local ones.

The approach in [21] is also based on a list scheduler and they consider the critical-path length, the critical resource usage and the register pressure in their heuristics. The critical-path length information is derived from both local and regional components.

Another difference with this work is in the choice of desirability factors: the factors taken into account are register pressure, speculation and list scheduling heuristics. In our work, the primary desirability factors are the PTD heuristic and the utilisation of concurrent resources. We will investigate in the future other possible factors that might contribute to improving the performance of the global scheduler.

3.1.5 Low complexity algorithms

Another goal in our research is to continue the line of low complexity time for the global scheduler. Moreno et al [22] use $\theta(n^2)$ complexity in their algorithms where necessary[4]. So far, the complexity of the algorithms for computing the dominator and post-dominator sets for each basic block in a region is $\theta(E)$, where E is the number of edges in the region; for the equivalent set, the complexity is $\theta(N)$ with N being the number of basic blocks[5].

3.1.6 Work to be done

- A global memory disambiguation mechanism. At present, we have assumed data dependencies for all memory references. It is apparent that these dependencies will limit the scope for movements not only locally but globally as well. We first have to implement a memory disambiguation mechanism in order to release these constraints.
- Functions to check-and-update data dependencies globally in an efficient manner.
- Implement the routines for the global scheduler. Integrate the previous points, the profile information and the PTD measure.
- Investigate if the PTD criteria is a good justification to speculate the movement of an instruction. We need very good reasons to justify the movement of an instruction speculatively without profile information.
- Modify the simulator in order to simulate traces and not to execute the instructions. This will allow us to simulate larger programs.
- We need further comparisons with the other approaches regarding register pressure and possibly with a global list scheduler.

3.1.7 Measuring the performance

The methodology to measure the effectiveness of our global scheduler can be viewed in two main phases: Firstly, as the global scheduler will be an extension of a local (PTD) scheduler, we will compare its performance improvement against the local scheduler. This position seems reasonable and can be justified

[4] This leaves open the question whether some algorithms may have higher complexity times.

[5] Given the binary-tree structure of the regions, we have at most $2N$ edges.

by the fact that work in the field usually assumes the same. Secondly, we can measure the global compilation overhead against the local time and then relate it with the span performance. This will give us an idea as to how costly the global transformations are.

We therefore propose performance comparisons between 1) the PTD local scheduler alone and the PTD scheduler with global code motion, and 2) between both list schedulers alone and with their counterparts with global motion implemented. We plan to compare as well 3) the compilation times between the PTD version (both local and global) and 4) the list schedulers (local and with code motion). We intend to have a clear view of the costs of these implementations in relation with their performances.

3.2 Global optimiser

The transformation of regions can be located in the optimisations that are independent of the target. By this we understand that it would benefit the performance of a program regardless of the nature of the architecture. However, among the common transformations that we can find in the literature, we will concentrate on those that specifically enable more ILP in a basic block term.

Although Figure 3 shows the need of number and type of resources in the architecture for the optimiser, these parameters are not strictly necessary. They can be used to help the optimiser to distribute the parallelism through the regions. In this section we discuss the issues around the global optimiser.

3.2.1 Related work

The trace scheduler [10][20] is a global optimisation technique that uses profile information to select a trace through the program. The trace is viewed as a block and it is scheduled with a list scheduler. Code motion is performed through the on-trace path at the expense of increasing compensation code in the off-trace. Sweany [29] proposes a variation of the trace scheduler to avoid the compensation code. The sperblock [14] is similar to trace scheduling with the difference that superblocks are single entry sub-graphs with no join points. All of these techniques use the list scheduler to optimise the code on-trace with appropriate heuristics which is the major difference to our work.

Region transformations are applied to programs in order to increase coarse and fine grain parallelism. The detection of coarse grain parallelism is usually performed with the source code, while for fine grain parallelism, the use of intermediate code or even assembly code is used. At present we are interested in fine grain or ILP optimisations that can be specifically helpful for scalar and superscalar machines.

The following region transformations are described in [12] and we will discuss each one with their relevances to an asynchronous architecture.

- τ_{unroll}

  Loop unrolling is a well known technique to increase the amount of ILP of loops. Our idea of unrolling n times a loop implies to create n copies of the body of the loop in order to have a new basic block with those n copies on it. This will effectively increase the number of possible concurrent instructions.

  However, as we see the loop unrolling definition in [12], the original n-iteration loop is transformed in two basic blocks, one being a m-iteration loop while the other being $n-m$ iteration as it can be seen in Figure 5 (a). This means that the new basic block has a side-entry from its tail. This definition has two differences with the above definition: first, the size of the basic block is constant, thus there are no differences (no advantages) if the loop is unrolled once or more times. Secondly, the new side-entry presumably would avoid the merger of another basic block. If the merger is done anyway it would introduce more executions of that basic block (see Figure 5 (b)). We believe that our definition may lead to better improvements than the last one.

  Loop unrolling is an optimising technique that distributes the parallelism of a program outside the loop. This gain can equally benefit both synchronous and asynchronous architectures.

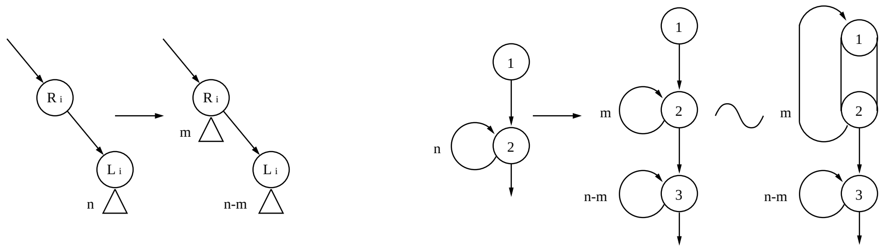

Figure 5: Unroll transformation: (a) shows a region (R_i) and a n-iteration loop (L_i) and their transformation, (b) shows the interpretation of this transformation. The third picture in (b) depicts the merger of blocks 1 and 2 inside the new loop. This might be a big execution overhead.

- τ_{invar}

  This transformation identifies code independent from the iterations of a loop (called *invariant code motion* elsewhere in the literature) which in practice means that the redundancies are removed away from the body of the loop. For our purposes, this kind of optimisations can be performed at a higher level for semantic reasons. It is more sensible to analyse the statements of a loop with source code than with assembly code. We can assume that this kind of redundance optimisations are already applied for us.

  Again, invariant code motion benefits synchronous or asynchronous targets in the same way as it removes the redundancies at a high level in the code.

- τ_{move}

  The τ_{move} transformation describes in general, code motion. Their definition of code motion does not respect speculation and unnecessary computations, and they argue that this does not lead to longer execution times. The justification is that this is only done if the parallelism in the destiny region has insufficient parallelism.

  Code motion of instructions or code motion of regions has to be done with some notion of the architecture. Although we are not sure if this notion has to imply a synchronous/asynchronous target, we believe that the coordination between the target and the optimiser is a matter of our concern. The notion could be simply as to the number and type of units in the architecture.

  At the moment we are interested in the code motion that covers the PTD heuristic and in the beginning we will look to code motion without speculation as described in the last section.

- τ_{copy}

  This transformation is the generalisation of *tail duplication* or *node splitting*. Tail duplication refers to the duplication of a node and its edges[6]. τ_{copy} basically duplicates more than one region.

  This transformation was originally applied to break cycles of dependences in order to generate better code for parallel machines as it helps to reduce communication and synchronisation costs [9]. Mueller and Whalley [23] propose *code replication* to avoid conditional branches. The resultant code contains simplified control flow that benefits vectorising and parallelising compilers.

  We can see that this transformation can lead to useful optimisations without any speculative nor unnecessary executions and its only drawback is code expansion.

- τ_{merge}

  The τ_{merge} transformation collapses two regions [7]. The equivalent for us would be two basic blocks under the same set of control dependencies (data sequentiality) that can be merged.

In the next section τ_{copy} and τ_{merge} are discussed in more detail with their relevance to our work.

3.2.2 Region transformation

Our idea of a global optimiser consists of transformations on basic blocks that are control-independent. As described in Section 3.1.3 the equivalence property

[6]A *tail* is a basic block with more than one entry edge to it

[7]Region in [12] represents parts of the program with the same set of control dependencies

is sufficient to determine the control independence between basic blocks. This is necessary to get a reduced representation as shown in Figure 4 (b), since in a control flow graph (CFG in 4 (a)) it is not trivial to know under which conditions a block is reached.

Among the different region transformations discussed in [12] for a LIW architecture, we find that τ_{copy} and τ_{merge} are useful for a scalar or superscalar processor because they do not introduce speculative executions.

In our representation, once the control dependencies have been sorted, the global optimiser can identify the nodes that are independent from the flow of control (blocks 4-6 and 5-6) to use tail duplication (see Figure 6 (a)). This enables another possible transformation that is the merger of the split nodes (6 and 6′) into their predecessors (4 and 5 respectively). Figure 6 (b) shows this transformation.

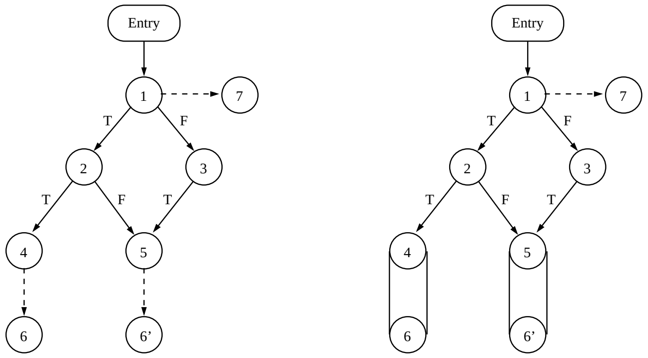

Figure 6: a) Tail duplication and b) node merger from the CFG in Figure 4

Although code duplication is not a good characteristic, the advantages of more ILP available by increasing the size of basic blocks are apparent. Furthermore, another point is that this possible improvement is independent of the different probabilities from the original paths (2-4-6 or 2-5-6), which in turn means that we do not need profile information to take decisions.

If we exhaustively perform tail duplication along the regions, the code expansion has an upper bound from the order of $\theta(p\, 2^p)$ where p is the number of *if-then-else* in the region.

3.2.3 Work to be done

- Implement the region transformation functions.
- Investigate the trade-offs between the cost of code expansion and the performance by joining basic blocks after tail duplication.
- Look at possible heuristics to determine when it is best to perform tail duplication.

- Investigate other possible useful region transformations that may suit asynchronous architectures.

4 A motivation example

We now present three benchmarks from the control-intensive set with the global optimisations hand-coded to test the potential of both techniques. Table 1 shows the performance improvements of a quick sort for four different optimisations: *Local* represents local optimisations only; *Tail* shows the transformation of regions by tail duplication; *Motion* takes into account the global code motion of instructions, and *T&M* represents the combination of *Tail* and *Motion* optimisations. Table 2 and Table 3 show the results for the Hanoi and integer matrix multiplication benchmarks respectively.

For these simulations we have compiled the benchmarks assuming infinite number of resources to eliminate the anti and output dependencies produced by performing register allocation before scheduling. Thus, the opportunities for global motion are increased. However, the the scheduling process has been done without memory disambiguation which restraints considerably the options for local scheduling after the movement of loads.

Conf.	Approach	SUIF	GM. sch.	Bal. sch.	PTD sch.	*
1 AU	Local*	0.00 %	11.89 %	11.86 %	16.73 %	0.00 %
	Tail	0.00 %	10.54 %	11.49 %	17.01 %	
	Motion	4.05 %	15.08 %	17.14 %	22.87 %	
	T&M	3.96 %	17.94 %	21.29 %	27.67 %	
2 AU	Local	0.00 %	23.31 %	23.41 %	23.24 %	15.05 %
	Tail	0.00 %	23.75 %	23.29 %	23.32 %	
	Motion	7.08 %	30.47 %	34.22 %	28.08 %	
	T&M	7.05 %	39.61 %	43.06 %	37.18 %	
3 AU	Local	0.00 %	27.99 %	29.51 %	25.20 %	15.74 %
	Tail	0.00 %	27.43 %	29.30 %	25.19 %	
	Motion	7.16 %	35.82 %	42.60 %	30.25 %	
	T&M	7.10 %	46.33 %	52.83 %	40.82 %	
4 AU	Local	0.00 %	30.29 %	31.29 %	25.04 %	15.75 %
	Tail	0.00 %	30.23 %	30.99 %	25.04 %	
	Motion	7.14 %	38.75 %	45.04 %	30.12 %	
	T&M	7.10 %	49.90 %	55.02 %	40.15 %	

Table 1: Performance comparison for the Bubble sort benchmark.

The tables are divided in to four groups representing simulations under four different architectures: 1 AU, 2 AU, 3 AU and 4 AU configurations. The percentage improvements in each block are all compared against the *Local* base case in that block. The star in the last column means the percentage improvement from each unscheduled code (SUIF) under the scaling architectures against the very base case (SUIF code under the 1 AU configuration).

Conf.	Approach	SUIF	GM. sch.	Bal. sch.	PTD sch.	*
1 AU	Local*	0.00 %	2.43 %	5.83 %	8.75 %	0.00 %
	Tail	0.00 %	2.43 %	5.81 %	8.75 %	
	Motion	0.01 %	2.42 %	5.79 %	8.71 %	
	T&M	0.01 %	2.42 %	5.79 %	8.71 %	
2 AU	Local	0.00 %	2.79 %	4.99 %	5.90 %	9.69 %
	Tail	0.00 %	2.79 %	4.98 %	5.90 %	
	Motion	0.01 %	2.77 %	4.96 %	5.92 %	
	T&M	0.01 %	2.77 %	4.96 %	5.92 %	
3 AU	Local	0.00 %	3.71 %	5.94 %	4.88 %	10.76 %
	Tail	0.00 %	3.71 %	5.94 %	6.79 %	
	Motion	0.01 %	3.72 %	5.91 %	6.84 %	
	T&M	0.01 %	3.72 %	5.91 %	6.84 %	
4 AU	Local	0.00 %	3.55 %	5.83 %	5.94 %	10.93 %
	Tail	0.00 %	3.55 %	5.84 %	6.90 %	
	Motion	0.01 %	3.50 %	5.81 %	6.94 %	
	T&M	0.01 %	3.50 %	5.81 %	6.94 %	

Table 2: Performance comparison for the Hanoi benchmark.

As it can be seen, the improvements from the *Tail* optimisations are basically the same to those achieved from the *Local* approach. This is due to the fact that the merger of a block to its tail introduces many memory-memory dependencies that the memory disambiguation mechanism has to remove. In the future this mechanism should enable better results.

In the other hand, the code motion performances show that this technique has a big impact over the local schedulers. Furthermore, this was done by moving only 17 instructions in the Bubble sort, 8 in the Hanoi benchmark and 10 instructions for the matrix multiplication one. If we accomplish our goal towards the complexity of the global scheduler, the overhead of performing these global transformations will be compensated by their improvements.

We have to say that the integer matrix multiplication benchmark does not have any "tail" nodes, so we could not apply the *Tail* duplication and merger optimisation, therefore the same results as in the *Local* approach.

One thing that we will carefully look is the scalability of the PTD scheduler. The Bubble sort benchmark experiences important improvements with the global optimisations when the architecture scales, but the PTD scheduler seems to loose part of this performance. This degradation does not appear when we apply scheduling after register allocation as it can be seen in the results in Appendix A.

Finally, the combination of global code motion and tail duplication ($G\&T$) exposes a scope for improvements that is the true motivation for performing these kind of optimisations.

We have seen during these transformations the importance of the decisions to apply tail duplication and code motion. For the former, we believe that it will be beneficial where the merger of blocks does not include any branches, which effectively means that the size of the block grows; for the latter, it will be decisive the use of the PTD measure to evaluate the benefits by moving an instruction. Among the 17 instructions that were moved in the Bubble sort benchmark, half of them were really useful.

Conf.	Approach	SUIF	GM. sch.	Bal. sch.	PTD sch.	*
1 AU	Local*	0.00 %	21.83 %	17.85 %	24.68 %	0.00 %
	Tail	0.00 %	21.83 %	17.85 %	24.68 %	
	Motion	1.19 %	22.12 %	18.18 %	25.33 %	
	T&M	1.19 %	22.12 %	18.18 %	25.33 %	
2 AU	Local	0.00 %	43.42 %	36.21 %	43.48 %	20.54 %
	Tail	0.00 %	43.42 %	36.21 %	43.48 %	
	Motion	1.83 %	44.78 %	37.75 %	44.02 %	
	T&M	1.83 %	44.78 %	37.75 %	44.02 %	
3 AU	Local	0.00 %	47.57 %	40.00 %	48.99 %	26.20 %
	Tail	0.00 %	47.57 %	40.00 %	48.99 %	
	Motion	1.99 %	48.45 %	41.62 %	50.18 %	
	T&M	1.99 %	48.45 %	41.62 %	50.18 %	
4 AU	Local	0.00 %	47.96 %	40.82 %	46.80 %	26.96 %
	Tail	0.00 %	47.96 %	40.82 %	46.80 %	
	Motion	1.96 %	49.07 %	42.38 %	47.85 %	
	T&M	1.96 %	49.07 %	42.38 %	47.85 %	

Table 3: Performance comparison for the integer matrix multiplication benchmark.

5 Thesis outline

The following is an early outline of how the Thesis should look like.

- Chapter 1. Introduction
  - MAP description (model)
  - Scheduling problem
- Chapter 2. Local scheduler
  - Early schedulers
  - List schedulers
    - Characteristics
    - Common heuristics
  - PTD scheduler
    - PTD heuristic (idea)
    - Algorithm
    - Complexity
    - Comparisons
    - Results
    - Future work (improvements)
    - Limitations
- Chapter 3. Global scheduler
  - Related work
    - Code motion
    - Trace scheduling
    - Software pipelining
    - Superblock scheduling
  - Definitions
  - Code motion
    - Requirements
      - Program representation
      - Memory disambiguation
      - Local scheduler (cooperation)
    - Approaches
      - Heuristics based on the PTD measure
      - Speculation and duplication
    - Algorithms complexity
  - Results
- Chapter 4. Global optimiser
  - Related work
    - Tail duplication
    - Loop unrolling
    - Function inlining
    - Region scheduling
  - Location of the global optimiser
  - Global optimiser
    - Tail duplication
    - Block merger
  - Results
  - Limitations ?
- Chapter 5. Conclusions and future work
  - Summary
  - Discussion
  - Future work (open questions)

References

[1] Arvind, Damal; Rebello, Vinod. *On the Performance Evaluation of Asynchronous Processor Architectures.* In Proc. 3rd. Int. Workshop on Modelling Analysis and Simulation of Computer and Telecommunication Systems MASCOTS'95, January 1995, pp. 100-105, IEEE Press.

[2] Arvind, Damal; Mullins, Robert. *Micronets: A Model for decentralising Control in Asynchronous Processor Architectures.* In 2nd. Working Conference on Asynchronous Design Methodologies. May 1995, pp. 190-199, IEEE Press.

[3] Arvind, Damal; Rebello, Vinod. *Static Scheduling of instructions on Micronet-based Asynchronous Processors.* In Proc. 2th. Int. Symp. on Advanced Research on Asynchronous Circuits and Systems ASYNC'96 March 1996, pp. 80-91, IEEE Press.

[4] Arvind, Damal; Sotelo-Salazar, Salvador. *Scheduling Instructions with Uncertain Latencies in Asynchronous Architectures.* In Proc. 3rd. Int. Euro-Par Conference. August 1997, pp. 771-778.

[5] Banerjia, Sanjeev; Havanki, William; Conte, Thomas. *Treegion Scheduling for Highly Parallel Processors.* In Proc. 3rd. Int. Euro-Par Conference. August 1997, pp. 1074-1078.

[6] Bernstein, David; Rodeh, Michael. *Global Instruction Scheduling for Superscalar Machines.* ACM SIGPLAN '91. Conf. on Programming Language Design and Implementation. June 1991, pp. 241-255.

[7] Click, Cliff. *Global Code Motion Global Value Numbering.* ACM SIGPLAN. Conf. Programming Language design and Implementation. June 1995, pp. 246-257.

[8] Feo, John. *An analysis of the computational and parallel complexity of the Livermore Loops.* Parallel Computing Vol. 7, 1988, pp. 163-185.

[9] Ferrante, Jeanne; Ottenstein, Karl; Warren, Joe. *The Program Dependence Graph and its Use in Optimization.* ACT Trans. on Pragramming Languages and Systems, Vol. 9, No. 3, July 1987, pp. 319-349.

[10] Fisher, Joseph. *Trace Scheduling: A Technique for Global Microcode Compaction.* IEEE Transactions on Computers. vol. C-30, no. 7, July 1981, pp. 478-490.

[11] Gibbons, Phillip; Muchnick, Steven. *Efficient Instruction Scheduling for a Pipelined Architecture.* ACM SIGPLAN Symp. on Compiler Construction, July 1986, pp. 11-16.

[12] Gupta, Rajiv; Soffa, Mary Lou. *Region Scheduling: An Approach for Detecting and Redistributing Parallelism.* IEEE Trans. on Software Engineering, Vol. 16. No. 4. April 1990, pp. 421-431.

[13] Hank, Richard; Hwu, Wen-mei; Rau, Ramakrishna. *Region-Based Compilation: An Introduction and Motivation.* Proc. 28th. Int. Symp. on Microarchitectures MICRO 28. 1995, pp. 158-168.

[14] Hwu, Wen-Mei et al. *The Superblock: An Effective Technique for VLIW and Superscalar Compilation.* The Journal of Supercomputing 7. 1993, pp. 229-248.

[15] Kerns, Daniel; Eggers, Susan. *Balanced Scheduling: Instruction Scheduling when Memory Latency is uncertain.* Transactions on computers. 1993, pp. 278-289.

[16] Knoop, Jens; Rüthing, Oliver; Steffen, Bernhard. *The Power of Assignment Motion.* ACM SIGPLAN. Conf. Programming Language design and Implementation. June 1995, pp. 233-245.

[17] Ko, Michael. *Instruction Scheduling for Micronet-Based Asynchronous Processors.* M.Sc. Project Report. 1995, University of Edinburgh, Department of Computer Science.

[18] Lam, Monica; Wilson, Robert; *Limits of Control Flow on Parallelism.* In Proc. 19th. Int. Symp. on Computer Architecture. May 1992, pp. 43-57.

[19] Lo, Jack; Eggers, Susan. *Improving Balanced Scheduling with Compiler Optimizations that Increase Instruction-Level Parallelism.* SIGPLAN Notices 1995, pp. 151-161.

[20] Lowney, Geoffrey et al. *The Multiflow Trace Scheduling Compiler.* The Journal of Supercomputing 7. 1993, pp. 51-142.

[21] Mahadevan, Uma; Ramakrishnan, Sridhar. *Instruction Scheduling over Regions: A Framework for Scheduling Across Basic Blocks.* pp. 419-434.

[22] Moreno, Jaime; Moudgill, Mayan; Miranda, R. *Architecture compiler and simulation of a tree-based VLIW processor.* IBM Research Report RC20495, Research Division, Computer Sciences/Mathematics. July 1996.

[23] Mueller, Frank; Whalley David. *Avoiding Conditional Branches by Code Replication.* In Proc. ACM SIGPLAN 95 PLDI, June 1995, pp. 56-66.

[24] Palem, Krishna; Simons, Barbara. *Scheduling Time-Critical Instructions on RISC Machines.* In Proc. Transactions on Programming Languages and Systems. September 1993, pp. 632-658.

[25] Proebsting, Todd; Fischer, Charles. *Linear-time, Optimal Code Scheduling for Delayed-Load Architectures.* SIGPLAN 1991, pp. 256-267.

[26] Rau, Ramakrishna; Fisher, Joseph. *Instruction-Level Parallel Processing: History, Overview, and Perspective.* Journal of Supercomputing 7, 1993, pp. 9-50.

[27] Rebello, Vinod. *On the Distribution of Control in Asynchronous Processor Architectures.* Ph.D. Thesis, Department of Computer Science, University of Edinburgh, November 1996.

[28] Sotelo-Salazar, Salvador. *Static Analysis and Scheduling for Asynchronous Processor Architectures.* Thesis Proposal. University of Edinburgh, Department of Computer Science. September 1996.

[29] Sweany, Philip. *Inter-block Code Motion without Copies.* Ph.D Thesis, Department of Computer Science, Colorado State University. Fall 1992.

[30] Theobald, Kevin; Gao, Guang; Hendren, Laurie. *On the Limits of Program Parallelism and its Smoothability.* ACAPS Technical Memo 40 McGill University, School of Computer Science.

[31] Ullman, John. *NP-complete scheduling problems.* Journal of Computer and Systems Sciences. October 1975, 384-393.

A Comparison of the schedulers

Benchmark	SUIF	GM. sch.	Bal. sch.	PTD sch.
Loop1	0.00 %	16.32 %	12.43 %	20.93 %
Loop2	0.00 %	25.49 %	30.26 %	30.72 %
Loop3	0.00 %	17.25 %	19.49 %	28.01 %
Loop4	0.00 %	13.47 %	21.84 %	27.55 %
Loop5	0.00 %	21.98 %	18.11 %	26.39 %
Loop6	0.00 %	21.52 %	23.39 %	26.84 %
Loop7	0.00 %	9.63 %	10.05 %	11.37 %
Loop8	0.00 %	13.26 %	13.45 %	15.07 %
Loop9	0.00 %	4.85 %	5.17 %	11.28 %
Loop10	0.00 %	23.03 %	22.99 %	29.74 %
Loop11	0.00 %	19.42 %	22.60 %	28.48 %
Loop12	0.00 %	19.40 %	22.59 %	28.45 %
Loop13	0.00 %	24.95 %	26.82 %	29.57 %
Loop14	0.00 %	21.41 %	22.80 %	29.90 %
Average	0.00 %	17.99 %	19.42 %	24.59 %
Perm	0.00 %	11.15 %	9.60 %	12.83 %
Heapsort	0.00 %	9.47 %	10.23 %	13.32 %
Clinpack	0.00 %	15.24 %	15.09 %	19.79 %
Hanoi	0.00 %	2.44 %	5.86 %	8.78 %
Quick	0.00 %	5.27 %	5.60 %	8.49 %
Bubble	0.00 %	14.70 %	12.95 %	15.54 %
Intmm	0.00 %	6.85 %	7.07 %	11.18 %
Queens	0.00 %	5.62 %	6.03 %	9.41 %
Subloops	0.00 %	23.62 %	23.00 %	27.45 %
Average	0.00 %	10.48 %	10.60 %	14.09 %
Total Average	0.00 %	15.06 %	15.97 %	20.48 %

Table 4: Performance comparison for the 1 AU configuration.

Benchmark	SUIF	GM. sch.	Bal. sch.	PTD sch.
Loop1	0.00 %	28.33 %	17.82 %	27.88 %
Loop2	0.00 %	50.82 %	49.95 %	50.96 %
Loop3	0.00 %	39.18 %	39.70 %	42.62 %
Loop4	0.00 %	31.21 %	34.07 %	38.53 %
Loop5	0.00 %	45.38 %	35.13 %	44.11 %
Loop6	0.00 %	43.03 %	42.04 %	43.18 %
Loop7	0.00 %	8.94 %	9.24 %	9.06 %
Loop8	0.00 %	20.77 %	18.81 %	19.38 %
Loop9	0.00 %	6.73 %	2.15 %	9.43 %
Loop10	0.00 %	50.82 %	49.73 %	48.57 %
Loop11	0.00 %	46.49 %	44.29 %	47.88 %
Loop12	0.00 %	46.44 %	44.32 %	47.94 %
Loop13	0.00 %	50.49 %	43.88 %	53.17 %
Loop14	0.00 %	36.70 %	37.73 %	39.77 %
Average	0.00 %	35.38 %	33.49 %	37.32 %
Perm	0.00 %	11.57 %	6.72 %	11.69 %
Heapsort	0.00 %	12.22 %	12.77 %	11.02 %
Clinpack	0.00 %	28.17 %	25.14 %	28.24 %
Hanoi	0.00 %	2.78 %	4.99 %	5.98 %
Quick	0.00 %	10.70 %	10.31 %	11.23 %
Bubble	0.00 %	22.71 %	23.26 %	23.21 %
Intmm	0.00 %	12.43 %	11.99 %	11.42 %
Queens	0.00 %	7.59 %	7.66 %	6.50 %
Subloops	0.00 %	52.88 %	43.49 %	50.57 %
Average	0.00 %	17.89 %	16.26 %	17.76 %
Total Average	0.00 %	28.54 %	26.75 %	29.67 %

Table 5: Performance comparison for the 2 AU configuration.

Benchmark	SUIF	GM. sch.	Bal. sch.	PTD sch.
Loop1	0.00 %	34.19 %	21.92 %	36.11 %
Loop2	0.00 %	58.86 %	54.21 %	61.51 %
Loop3	0.00 %	48.01 %	52.83 %	50.92 %
Loop4	0.00 %	40.03 %	39.62 %	48.49 %
Loop5	0.00 %	50.81 %	37.47 %	52.64 %
Loop6	0.00 %	44.70 %	40.78 %	47.52 %
Loop7	0.00 %	8.67 %	8.92 %	8.70 %
Loop8	0.00 %	21.69 %	20.09 %	20.61 %
Loop9	0.00 %	7.18 %	5.62 %	7.29 %
Loop10	0.00 %	53.91 %	50.13 %	52.71 %
Loop11	0.00 %	54.43 %	50.27 %	57.41 %
Loop12	0.00 %	54.38 %	50.26 %	57.34 %
Loop13	0.00 %	63.21 %	57.44 %	67.91 %
Loop14	0.00 %	38.47 %	38.94 %	40.62 %
Average	0.00 %	41.32 %	37.75 %	43.56 %
Perm	0.00 %	12.25 %	7.22 %	11.85 %
Heapsort	0.00 %	11.30 %	11.89 %	9.85 %
Clinpack	0.00 %	27.14 %	25.25 %	25.30 %
Hanoi	0.00 %	3.74 %	5.95 %	6.85 %
Quick	0.00 %	9.90 %	9.95 %	9.99 %
Bubble	0.00 %	26.72 %	27.76 %	28.00 %
Intmm	0.00 %	11.51 %	10.74 %	10.40 %
Queens	0.00 %	6.67 %	6.74 %	5.51 %
Subloops	0.00 %	61.87 %	52.71 %	61.61 %
Average	0.00 %	19.01 %	17.58 %	18.82 %
Total Average	0.00 %	32.59 %	29.86 %	33.88 %

Table 6: Performance comparison for the 3 AU configuration.

Benchmark	SUIF	GM. sch.	Bal. sch.	PTD sch.
Loop1	0.00 %	34.02 %	21.73 %	37.15 %
Loop2	0.00 %	59.29 %	55.74 %	63.29 %
Loop3	0.00 %	48.43 %	44.52 %	54.54 %
Loop4	0.00 %	39.82 %	40.72 %	49.24 %
Loop5	0.00 %	50.94 %	37.30 %	53.87 %
Loop6	0.00 %	44.94 %	41.66 %	48.17 %
Loop7	0.00 %	8.87 %	8.95 %	8.73 %
Loop8	0.00 %	21.61 %	19.98 %	20.89 %
Loop9	0.00 %	6.90 %	5.45 %	7.26 %
Loop10	0.00 %	53.69 %	51.14 %	51.50 %
Loop11	0.00 %	54.78 %	50.60 %	58.53 %
Loop12	0.00 %	54.90 %	50.65 %	58.72 %
Loop13	0.00 %	62.20 %	57.48 %	67.24 %
Loop14	0.00 %	38.86 %	38.73 %	40.20 %
Average	0.00 %	41.38 %	37.48 %	44.23 %
Perm	0.00 %	13.52 %	7.03 %	13.07 %
Heapsort	0.00 %	11.17 %	11.82 %	9.91 %
Clinpack	0.00 %	26.64 %	24.87 %	25.25 %
Hanoi	0.00 %	3.56 %	5.87 %	6.94 %
Quick	0.00 %	9.69 %	9.77 %	9.99 %
Bubble	0.00 %	28.67 %	28.83 %	27.70 %
Intmm	0.00 %	11.27 %	10.60 %	10.21 %
Queens	0.00 %	6.62 %	6.69 %	5.57 %
Subloops	0.00 %	62.44 %	53.24 %	61.51 %
Average	0.00 %	19.29 %	17.64 %	18.91 %
Total Average	0.00 %	32.73 %	29.71 %	34.33 %

Table 7: Performance comparison for the 4 AU configuration.