# Automatic Parallelization in the Paralax Compiler

Hans Vandierendonck
Ghent University
Dept. of Electronics and Information Systems
Sint-Pietersnieuwstraat 41
9000 Gent, Belgium
hvdieren@elis.ugent.be

Koen De Bosschere
Ghent University
Dept. of Electronics and Information Systems
Sint-Pietersnieuwstraat 41
9000 Gent, Belgium
kdb@elis.ugent.be

## ABSTRACT

The efficient development of multi-threaded software has, for many years, been an unsolved problem in computer science. Finding a solution to this problem has become urgent with the advent of multi-core processors. Furthermore, the problem has become more complicated because multi-cores are everywhere (desktop, laptop, embedded system). As such, they execute generic programs which exhibit very different characteristics than the scientific applications that have been the focus of parallel computing in the past.

Implicitly parallel programming is an approach to parallel programming that promises high productivity and efficiency and rules out synchronization errors and race conditions by design. There are two main ingredients to implicitly parallel programming: (i) a conventional sequential programming language that is extended with annotations that describe the semantics of the program and (ii) an automatic parallelizing compiler that uses the annotations to increase the degree of parallelization.

It is extremely important that the annotations and the automatic parallelizing compiler are designed with the target application domain in mind. In this paper, we discuss the Paralax approach to implicitly parallel programming and we review how the annotations and the compiler design help to successfully parallelize generic programs. We evaluate Paralax on SPECint benchmarks, which are a model for such programs, and demonstrate scalable speedups, up to a factor of 6 on 8 cores.

## Categories and Subject Descriptors

Software [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; Software [**Programming Languages**]: Processors—*Compilers*

## General Terms

Algorithms, Design, Performance

## Keywords

Automatic parallelization, semantic annotations


SCOPES '11, June 27-28, 2011, St. Goar, Germany


## 1. INTRODUCTION

Programming multi-core processors is an error-prone and time-consuming process because the most popular parallel programming model exposes very low-level primitives to the programmer, such as threads and synchronization primitives. This situation potentially constitutes a new software crisis [1] because this model does not allow us to abstract away parallelism from everyday programming. Parallelism affects all parts of software, even those that are intended to execute serially.

Several current research projects aim to develop parallel programming models that are less error-prone and less time-consuming to develop programs in. One such approach are implicitly parallel programming models [6, 24], where it is assumed that sequentially specified programs *implicitly* contain parallelism and that this implicit parallelism can be *automatically* extracted. For a program to contain implicit parallelism, it is very important that the programmer cooperates by selecting parallelizable algorithms and, potentially, also by adding annotations to the program that describe higher-order semantic properties. This implicit parallelism is then extracted using an automatic parallelizing compiler.

There are many advantages to implicitly parallel programming models: the use of a sequential language implies that we can reuse all tools and methodologies to develop, to reason about and to debug sequential programs. Furthermore, the explicit use of threads and synchronization primitives is hidden in the compiler and the runtime system for which it generates code. As such, the error-prone part of parallel programming is taken care of by tools and typical bugs such as dead-lock and race conditions have to be solved only once, when developing the tools.

Implicitly parallel programming however depends on successful automatic parallelizing compilers. Automatic parallelization has been successful especially in the context of scientific computations, but success for generic computations has been largely lacking. There are two major reasons why automatic parallelization of implicitly parallel programs can be successful where automatic parallelization in general falls short. First, we require that programs are parallelizable, which means that failure to parallelize is not a failure on the part of the compiler, but it is an expected outcome given the input. Second, implicitly parallel programs are enriched with semantic annotations that can significantly increase the precision of compiler analysis. For instance, annotations may summarize memory side effects of procedures, implying that memory dependency analysis across call sites becomes much more precise.

Generic programs have a rich code structure, which poses many challenges to auto-parallelizing compilers. Inner loops in these programs are small and iterate few times [9], so parallelism must be sought in the outermost loops. These programs typically do not contain important DOALL loops. Parallel loops are various sorts

of pipelines [3]. Furthermore, the parallel loops contain system calls, which must be kept strictly ordered [8].

In this paper, we discuss the design of the Paralax compiler [24], a compiler for automatic parallelization of implicitly parallel programs. The Paralax compiler recognizes several annotations in the program and aims at extracting various types of pipeline parallelism. Furthermore, it performs state-of-the-art memory analysis and performs inter-procedural code restructuring to expose parallelism. We demonstrate the Paralax compiler by parallelizing implicitly parallel versions of several SPECint benchmarks.

In the remainder of this paper, we first discuss alternative approaches to averting a new software crisis (Section 2) and we present the Paralax approach to parallel programming (Section 3). Next, we give a bird's eye overview on the design of the Paralax compiler and discuss the identification of parallelism (Section 4), preprocessing of the program representation (Section 5) and the parallel code generation (Section 6). Then, we evaluate the Paralax compiler on SPECint benchmarks (Section 7) and we conclude the paper (Section 8).

## 2. RELATED WORK

Numerous current research projects aim to improve the programmability of multi-core processors. We discuss some of the most similar to the presented work.

The implicitly parallel programming approach was described by Hwu *et al.* [6]. Several research projects focusing on automatic parallelization in essence follow these ideas too.

David August's group has contributed many automatic parallelization techniques for various types of pipeline parallelism [12, 15, 20]. In this work, a sequential programming model is also extended with annotations [3]. Speculation is another essential ingredient [14] and is used to avoid complex memory disambiguation and to break dependencies in loops. As a consequence, non-speculative parallelism also takes the overhead of speculation.

Other parallelization strategies aim to parallelize largely the same loops but utilize different analysis techniques and code generation schemes. Copy-or-discard [18] identifies memory objects used in parallel loops and speculatively privatizes them. In [25], pipelined loops are mapped onto ordered transactions.

Still other techniques rely in part on directives inserted by the programmer. Directives are quite different from annotations as they direct the compiler to take certain actions, while the annotations used in Paralax do not directly describe parallelization. Software behavior oriented parallelization [4] speculatively parallelizes user-identified loops. This paper was the first to show that speculation can be efficiently implemented by utilizing multiple processes and page sharing between processes. Thies *et al.* [17] require the programmer to identify pipeline stages in loops.

A quite distinct approach is followed by Deterministic Parallel Java [2]. Here, the Java type system is extended with effects that describe disjointness of memory regions. Hereby, it is possible to *type-check* the correctness of parallel constructs in a program.

Task-based languages share many of the benefits of implicitly parallel languages. In a task-based language, the programmer identifies tasks together with their memory footprints. A runtime system analyzes inter-task dependencies and schedules independent tasks concurrently, much like an out-of-order superscalar processor executes instructions [13]. Furthermore, it can be proved that, for a particular language, all parallel executions of a task-based program compute exactly the same value as the serial execution of the program [22]. In [7], tasks are identified by the programmer while task memory footprints are inferred by the compiler, relieving the programmer of this task.

**Table 1: Annotations recognized by Paralax**

| Annotation | Semantics |
|---|---|
| Function arguments | |
| MOD | Pointed-to memory is modified |
| REF | Pointed-to memory is referenced |
| KILL | Pointed-to memory is invalidated |
| NOCAPTURE | Pointer is not captured (doesn't escape) |
| RETALIAS(#) | Return value is pointer and aliases argument number # |
| Functions | |
| SYSCALL | Function may have externally visible side-effects |
| STATELESS | Function does not maintain internal state |
| COMMUTATIVE | Function is commutative |
| CONSTRUCTOR(fn) | Function is a constructor, *fn* is the corresponding destructor |
| Variables | |
| KILL(var) | Statement specifying that *var* is dead |

## 3. THE PARALAX APPROACH TO PARALLEL PROGRAMMING

The Paralax approach to parallel programming is an implicitly parallel approach. As such, it provides semantic annotations on top of C/C++ as well as an auto-parallelizing compiler. Furthermore, Paralax provides a dynamic analysis tool that proposes program locations to the programmer where annotations could improve auto-parallelization.

### 3.1 Annotations

Table 1 specifies the set of annotations currently recognized by Paralax. Annotations apply to procedure arguments, to procedures or to program variables. A detailed specification of these annotations can be found in [24].

The procedure argument annotations are quite straightforward. The RETALIAS annotation is somewhat special: it states that the return value is a pointer that is aliased to one of the procedure arguments. This behavior occurs frequently in the C library and it is important to notify the compiler of it.

The procedure annotations specify overall procedure behavior. A COMMUTATIVE procedure may be reordered with respect to itself [3], e.g. a memory allocation procedure or a procedure that inserts an element in a set. Commutativity can increase parallelism. A STATELESS procedure does not read/write memory except through pointers passed as arguments.[1] The CONSTRUCTOR annotation declares a constructor/destructor pair of procedures. The constructor allocates memory and potentially initializes it. The destructor performs cleanup actions and deallocates the memory. Spotting this behavior is helpful for the compiler. E.g. the compiler knows that the constructor returns a "fresh" pointer that is not aliased to anything else.

The KILL statement declares that a memory object is *dead* at a particular point in the program. This means that we will not use the contents of the object anymore. As such, the compiler can safely privatize the object (create distinct per-thread copies) [19]. The KILL statement is very powerful to expose coarse-grain parallelism in programs that recycle memory objects in each loop iteration.

[1]Note that GCC defines a pure function as a function that depends only on its arguments and global variables. A pure function does not modify memory. As such, the STATELESS functions are a superset of GCC's pure functions.

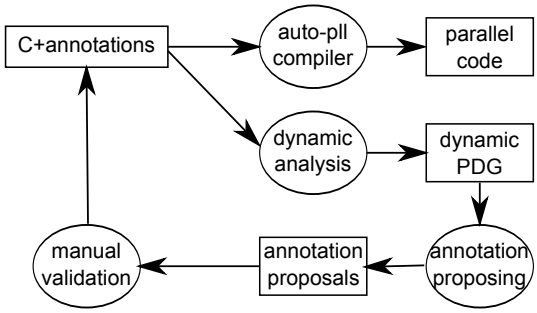



**Figure 1: Paralax tool flow.**

## 3.2 Proposing Annotations

Sometimes it may be hard for a programmer to understand what annotations are missing from a program to enable auto-parallelization. For this reason, Paralax provides a dynamic analysis tool [16] that measures all memory dependencies as they occur during a particular program execution. This dynamic memory dependency information allows us to propose locations for inserting the KILL annotations. Hereto, we compute the difference between the dynamic memory dependency information and the static memory dependency information computed by the compiler. Any dependency edge that is assumed by the compiler but does not occur in the dynamic information is potentially due to conservatism or lack of precisions of the compiler. From these dependencies, we take those where (i) a KILL annotation would be valid and (ii) parallelism increases after inserting the annotation. The detailed algorithm is presented in [24].

## 3.3 Running Example

Listing 1 shows an example program where the procedure *p* contains a parallel loop. *p* calls on procedures *f* and *g*. Note how the annotations are used to describe side-effects of the procedures. None of the procedures modifies the memory pointed to by its argument, so this argument is annotated as REF (read). Furthermore, the argument is not stored to memory, so it is also annotated as NOCAPTURE. Finally, *g* is a STATELESS procedure, implying that it only accesses memory through its argument. This property is clearly not true for *f*, which accesses the global variable *sum*.

Note that one might also annotate *f* as a COMMUTATIVE procedure in this calling context because *g* is a pure procedure (read-only and STATELESS) and because the accumulation in sum is commutative (modulo floating-point rounding errors). In this case, this would not help us to obtain scalable parallelism because we can still execute only one instance of *f* at a time. We will show how the Paralax compiler transforms the code to execute multiple instances of *f* concurrently.

# 4. PARALLELIZATION IN THE PARALAX COMPILER

The Paralax compiler is implemented on top of LLVM[2] as a bytecode to bytecode transformation.

## 4.1 Overview of the Paralax Compiler

Listing 2 shows an overview of the major parallelization steps. The compilation process starts with data structure analysis [10] (DSA), a unification-based shape analysis that recovers the identity and type (e.g. array, structure) of all memory objects by inspecting the instructions that access them. DSA also identifies when one object holds a pointer to another one and indicates when pointers to objects may escape or when objects are aliased to other objects.

DSA describes the memory objects that are used in a program. Sometimes this information is very precise, but it may happen that this information is imprecise due to limitations in the analysis, external procedure calls, etc. In those cases, DSA will flag this event, which basically limits us to manipulate the memory object in any way. For this reason, we divide the memory objects in two sets: the copyable objects are fully characterized while the non-copyable objects have been flagged by DSA as incompletely analyzed.

[2] http://www.llvm.org/

**Listing 1: Example program in C**

```
float sum = 0;
float g(struct node * REF NOCAPTURE
        ptr) STATELESS; /* pure */
void f(struct node * REF NOCAPTURE ptr) {
    float z = g(ptr);
    sum = sum + z;
}
void p(struct node * REF NOCAPTURE ptr) {
    while(ptr) {
        if(ptr->good)
            f(ptr);
        ptr = ptr->next;
    }
}
```

**Listing 2: Paralax compilation flow**

```
algorithm parallelize
begin
  analyze memory objects (DSA)
  foreach loop in loops do
  begin
      PDG = program dependence graph(loop)
      DAG_SCC = strongly connected
                components(PDG)
      // PDG and DAG_SCC pre-processing
      pre-process DAG_SCC
      while a callee has cyclic dependencies
            on non-copyable objects do
      begin
          extract ordered sections from callee
          recompute PDG and DAG_SCC
      end
      if |DAG_SCC| = 1 next loop
      // Parallelization
      foreach partitioning method P do
          apply P to DAG_SCC
          estimate speedup, remember best P
      done
      if best speedup <= 1 next loop
      // Code generation
      apply parallelization strategy of
        best partitioner P:
      begin
          apply privatization
          determine communicated values
          generate multi-threaded code
          rewrite callees
      end
  end
end
```

## 4.2 Parallelization

Parallelization follows largely decoupled software pipelining [12, 15] (DSWP). Parallelization is performed by analyis of the program dependence graph [5]. The PDG captures all dependencies in a pro-

**Listing 3: Assembly code for the example procedure *p***

```
1  H:  r2 := cmp r1, 0
2      brz r2, X
3  B:  r2 := M[r1+8]
4      r3 := cmp r2, 0
5      brz r3, L
6      call f, r1
7  L:  r1 := M[r1]
8      br H
9  X:  /* ... */
```

gram, including data dependencies (mediated by "register" values), control dependencies and memory dependencies.

The PDG may contain cycles. For instance, the loop termination test is part of a cycle and dependencies that span multiple loop iterations also manifest themselves as a cycle.

Cycles must always be contained within a single thread to minimize inter-thread communication. Hence, we compute all strongly connected components on the PDG and build the directed acyclic graph of strongly connected components (DAG-SCC). Using the DAG-SCC, we can detect various parallel loop patterns such as pipelines, parallel-stage pipelines and DOALL loops, as well as a parallel sections construct in non-loop code.

Listing 3 shows the assembly code for procedure $p$[3] and Figure 2 shows the corresponding program dependence graph along with the DAG-SCC. The PDG contains several cycles. The loop termination branch is part of a cycle with the loop termination condition and with the update of the loop iteration variable. Also, node 6, which corresponds to the call to *f*, has a dependency on itself because of the update of the global variable sum. These cycles in the PDG imply that the modeled loop is not a DOALL loop and that a more restrained form of parallelism must be used. These types of parallelism are discussed next.

## 4.3 Parallelization Using the DAG-SCC

Pipeline parallelism is extracted from the DAG-SCC by splitting the graph in pipeline stages. The pipeline stages are a partition of the nodes of the DAG-SCC. Pipeline stages are ordered: no edge in the DAG-SCC points from a pipeline stage to a previous one.

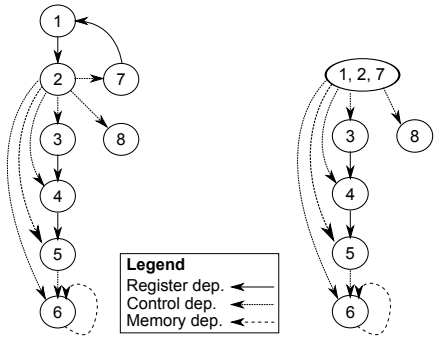
DAG-SCC splitting allows to exploit pipeline parallelism only. To exploit data parallelism, we differentiate between cyclic and acyclic DAG-SCC nodes. Cyclic nodes contain a subset of the PDG nodes where a cycle of edges runs through the PDG nodes. E.g. the strongly connected component 1-2-7 and node 6 in Figure 2 contain cycles. All other nodes in the example do not contain cycles and are acyclic.

Note that the loop control is always a cyclic node that contains at least the code that determines loop termination. In the case of counted loops (e.g. `for i=1 to N` where `N` is a loop constant), this SCC can be parallelized because the loop iteration range can be properly divided between the threads. As such the loop control SCC of counted loops must not be considered as a cyclic node. This potentially increases the degree of parallelism in the loop.

## 4.4 Recognized Parallel Patterns

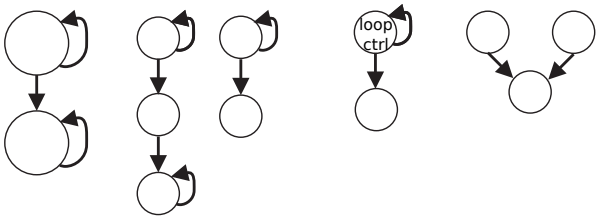
Four parallel patterns are recognized by the compiler. General properties of the program dependence graphs are depicted in Figure 3. We discuss them one by one.

[3]Although the Paralax compiler uses LLVM bytecodes internally instead of assembly code, we show assembly code here because it is more well-known.




(a) PDG (b) DAG-SCC

**Figure 2: PDG of procedure *p* in Listing 3. Numbers in the node correspond to statement numbers in the code listing.**




(a) Pipeline (b) Parallel-stage pipelines (c) DOALL loop (d) Parallel sections

**Figure 3: Depiction of program dependence graphs properties for the recognized parallel patterns.**

### *Pipelines.*

A pipeline consists of a succession of pipeline stages where each stage is a collection of potentially cyclic DAG-SCC nodes.

The pipeline must be well balanced, i.e. each stage must execute approximately the same amount of work. An algorithm that balances a pipeline over a pre-determined number of pipeline stages is described in [12]. Because the execution time of a pipeline also depends on the delay incurred by communication, the optimal number of pipeline stages is not necessarily equal to the maximum number of threads. Hence, we repeat this algorithm for 2 threads up to the maximum number of threads and we select the number of pipeline stages where estimated performance is optimal.

Pipelines allow parallelism to scale to the number of pipeline stages at best. As such, parallelism is limited by the code structure, not by the data set size.

### *Parallel-Stage Pipelines.*

Parallel-stage pipelines contain pipeline stages with only acyclic nodes that consume an important fraction of the execution time of the loop. Such pipeline stages are special as it is allowed to execute instances of the stage corresponding to different loop iterations concurrently (data parallelism). As such, parallelism scales much further than the number of pipeline stages. Parallelism is basically bound by the fraction of time spent in the parallel stages compared to the time spent in the whole loop (Amdahl's Law).

Parallel-stage pipelines are constructed differently than regular pipelines: instead of evenly balancing the work between pipeline stages, it is now attempted to put as much work as possible in the parallel stages [15]. Furthermore, it is best to assign one thread to each of the sequential stages and to assign all other available threads to execute instances of the parallel stage.

Parallelism scales with the data set size when concurrently executing multiple instances of the parallel stage. Hereby, parallel-stage pipelines are a major source of scalable parallelism.

*DOALL Loops.*

DOALL loops are characterized by the absence of cyclic nodes in the DAG-SCC. Note that the treatment of the loop control SCC is especially important in the case of DOALL loops: cycles on the loop control SCC must not be considered.

*Parallel Sections.*

Parallel sections are non-loop parallel constructs that consist of two or more time-consuming code fragments that may execute in parallel. These code fragments are computed from the DAG-SCC by identifying two or more nodes where neither node is dependent on the other and either all nodes execute or none of them execute (they are control equivalent). This definition corresponds closely to the definition of parallel sections in OpenMP.

# 5. PDG AND DAG-SCC PREPROCESSING

Before executing the loop pattern matchers, we perform a number of restructurings of the PDG and/or the DAG-SCC that are either necessary for correctness or that expose additional parallelism.

## 5.1 Non-Copyable Memory Objects

We distinguish between copyable and non-copyable objects, where the latter are objects (or memory accesses) that data structure analysis fails to analyze with sufficient accuracy. Such objects must be treated with care during analysis and program transformation. In particular, we cannot safely duplicate or copy such objects.

Similar conservatism must be applied when pointers to an object may be loaded from memory during a parallelized loop. Translating such pointers to privatized copies of an object is non-trivial. We side-step this problem by treating such objects as non-copyable.

We restrict the parallelization of loops such that all accesses to non-copyable objects are performed from within the same pipeline stage. Under this condition, the parallel code generation will never try to duplicate such an object.

Accesses to a non-copyable object are forced in the same pipeline stage by merging all nodes in the DAG-SCC that operate on the same non-copyable object. The resulting node is acyclic when each of the merged DAG-SCC nodes was acyclic. This allows parallel execution of the resulting node.

## 5.2 Inner Loops

In the Paralax compiler, an inner loop always appears as a cycle in the DAG-SCC. This is in contrast to the DSWP work where an inner loop may be split across pipeline stages [12]. The main motivation for this choice is that (i) we can largely emulate this idiom by means of the ordered sections optimization discussed below, and (ii) it simplifies the communication of values between pipeline stages. Thus, we artificially merge all nodes corresponding to the same inner loop before analyzing parallelism.

The merged inner loop SCC must be correctly labeled as cyclic or acyclic. An acyclic SCC would allow multiple instances of the inner loop to execute in parallel. We model this condition by relabeling all nodes as acyclic in the inner loop before merging the nodes, except for those nodes that modify non-copyable objects. If the merged loop SCC is now cyclic, then the cycle must include instructions that belong to the outer loop (or the loop modifies non-copyable objects), preventing parallel execution of multiple instances of the inner loop.

## 5.3 Ordered Sections in Callees

Paralax implements a mechanism to increase parallelism in loops that contain procedure calls with *ordered sections* [23]. An ordered section in a callee procedure is a fragment of the callee PDG where memory objects are updated but where the remainder of the PDG does not depend on the updated objects. The fragment of the PDG is thus a sink for memory dependencies.

The problem with ordered sections is that they can introduce cross-iteration dependencies in the calling loop, prohibiting parallel execution of the callee (either parallel to other calls of the same procedure or parallel to other code in the loop). Ordered sections occur frequently: print statements, profiling counters and system calls inhibit parallelism, but factoring out these code sequences exposes parallelism.

The definition of procedure $f$ in the running example (Listing 1) has a computational part that does not modify memory and an update of the global object *hist*. The update of *hist* introduces a cyclic dependency on the call to $f$, strictly ordering its executions. The cyclic dependency can however be easily removed by not performing the update of the global memory object as part of $f$. Therefore, the Paralax compiler tries to move the reduction on the *sum* variable from $f$ to the loop body.

Ordered sections are characterized by their *update set* (the set of objects that it modifies), their *copy set* (the set of register variables and objects that it only reads) and the *factored code*. The factored code is a fragment of the DAG-SCC of the containing procedure that includes all instructions that reference any of the objects in the update set as well as all of their dependents.

In the running example, the update set of the ordered section is *hist*, the read set is the temporary $z$ and the factored code is the statement *sum = sum + z;*. Note that this information is sufficient to move the factored code to a different location in the program.

The update set of an ordered section is found by analysis of the DAG-SCC of the parallelized loop. We use cycles of edges in the DAG-SCC that are built from memory dependencies on one particular procedure object as seeds for the update sets. The procedure calls in the DAG-SCC node are then recursively analyzed to identify the ordered sections. The PDG is then modified by splitting the procedure call nodes in two separate nodes: one representing the main body of the procedure and one representing the execution of all ordered sections in this procedure call. Edges in the PDG are modified such that all memory dependency edges on the update set of the ordered section are redirected to the ordered section node of the call. Furthermore, a dependency is added from the main body of the call to the ordered section node to indicate the causal relationship. The DAG-SCC is then recomputed and new ordered sections are searched for by repeating the algorithm above.

For the example (Figure 2), identification of the ordered section allows to split node 6 in a node for the main body of $f$ (node 6-MB) and a node for the ordered sections (node 6-OS) (Figure 4). Node 6-MB is now acyclic, which is a plus for parallelization as ordered sections typically consume less time of procedure calls. After the transformation, the code can be parallelized using the more scalable parallel-stage pipeline instead of a regular pipeline.

Besides analyzing ordered sections in callees, we also try to factor out ordered sections from inner loops, because the inner loops are represented by a single undivisable node in the DAG-SCC, just like procedure calls. The mechanisms to achieve this are principally the same as for the procedure call case.

# 6. PARALLEL CODE GENERATION

## 6.1 Privatization

Privatization is a code transformation where a fresh copy of a memory object is allocated for each thread when multiple threads may be working on this object [19]. In the context of pipeline parallelism, this may occur in one of two situations: (i) the object is

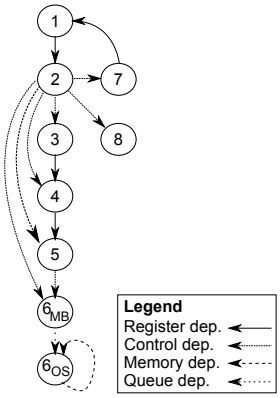



**Figure 4: Program dependence graph for the code of Listing 1, assuming that ordered sections have been removed from *f*.**

read and written by a parallel stage and (ii) the object is read by multiple pipeline stages and written by at least one. In this cases, a fresh copy of the object is allocated for each loop iteration. We privatize objects only when they are completely overwritten (killed), which greatly simplifies computing the loop live-out value [21].

## 6.2 Communicating Values

When code is split across multiple threads, some values need to be sent from one thread to the other. Hereto, the DSWP algorithm assumes the existence of queues (FIFOs) between the threads and two primitives [12]:[4] *produce*, to produce a value in a queue and *consume*, to consume a value from a queue. The DSWP algorithm defines where to place produce and consume operations in order to have a correct execution of the parallel program.

While DSWP communicates every value separately, Paralax assumes a lumped communication model where all values produced in a pipeline stage are sent at once at the end of the stage and they are all received at once at the start of the stage. Lumped communication is effected by defining for each parallelized loop a *communication structure*. This is a structure in the C sense of the word that contains all values that are communicated between pipeline stages. One instance of this structure is created per loop iteration. The produce and consume primitives now translate to loads and stores from the communication structure.

The values that must be communicated essentially correspond to the edges of the PDG that have their head and tail in different pipeline stages. These values can be register values (corresponding to data dependencies), branch directions (corresponding to control dependencies) and privatized memory objects. Each such value is assigned a slot in the communication structure. Privatized objects may be embedded in the structure when the size of the object can be determined at compile-time. Otherwise, the communication structure contains a pointer to dynamically allocated memory.

Adopting a lumped communication approach has several consequences. On the one hand, the number of dynamically executed produce/consume operations is much less in our approach, even when optimizing the communication [11]. On the other hand, there is no overlap in the execution of dependent pipeline stages because the second pipeline stage can start an iteration (receive values) only when the first has finished an iteration (send values). In contrast, the DSWP approach allows partial overlap of dependent pipeline

[4]In the original work, it was assumed that these queues are implemented in hardware, allowing fast inter-processor communication. Paralax implements queues in software.

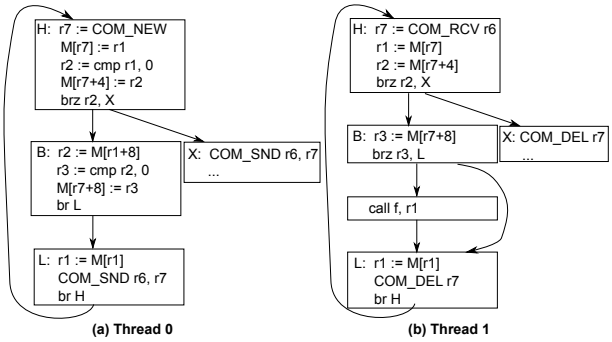



**Figure 5: Control flow graphs for the parallelized code of Listing 3, assuming parallelization as a 2-stage pipeline.**

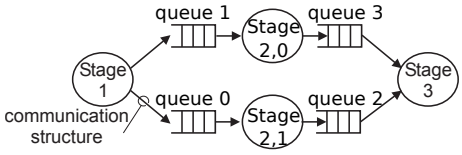



**Figure 7: Thread mapping and queue numbering for a 3-stage parallel-stage pipeline using a total of 4 threads.**

stages. We feel however that this has little importance, because the underlying idea of DSWP is to pipeline the instructions in a loop body in such a way that communication latencies can be tolerated. Postponing the communication of values should therefore have little effect on performance.

## 6.3 Threaded Code Generation

Code is split across multiple threads using the multi-threaded code generation algorithm of DSWP [12]. This algorithm determines what basic blocks must be replicated in each thread, which control transfers must be replicated and when values must be sent and received. Figure 5 shows the control flow graphs for the program of Listing 3 assuming that it is parallelized as a 2-stage pipeline. As such there are 2 threads: a producing thread and a consuming thread. Note that the instructions of the original program in principle occur in only one of the 2 threads. Branch instructions are an exception as the control flow must be equal in both threads.

The commands `COM_NEW` and `COM_DEL` create and delete a new communication structure, respectively. The commands `COM_SEND` and `COM_RCV` send and receive, respectively, a pointer to such a structure over the queue connecting two threads, which is in our case identified by a memory address stored in register r6.

The fields of the communication structure are initialized in the producing thread by regular memory store instructions. In the consuming thread, load instructions restore the values of registers. These loads and stores are located at the positions where DSWP inserts consume and produce statements, respectively.

Generating code for parallel-stage pipelines is somewhat more involved. We illustrate here the parallelization of Listing 1 over 4 threads, i.e. the parallel pipeline stage is replicated twice. Figure 6 shows the control flow graphs for each thread. Figure 7 shows how pipeline stages are mapped to threads and what queues are used for inter-thread communication. We assume that the first queue is identified by register r6 and that following queues are identified as r6+1, r6+2 and r6+3 as a matter of convenience. In reality, less readable address calculations are inserted in the code.

Thread 0, implementing pipeline stage 1, statically divides the loop iterations between the threads for stage 2. In particular, a reg-

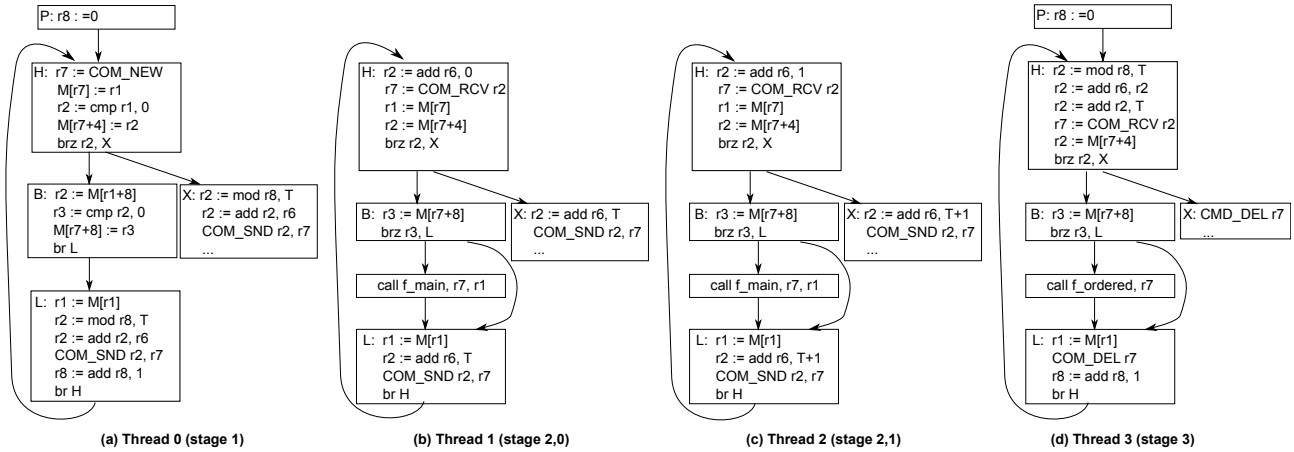



**Figure 6: Control flow graphs for the parallelized code of Listing 1, assuming parallelization as a 3-stage parallel-stage pipeline.**

ister r8 is incremented on each loop iteration and when r8 is even, the communication structure is sent to queue 0, otherwise it is sent to queue 1. A similar modification occurs in the final pipeline stage 3, where the communication structure is taken from queue 2 on even loop iterations and it is taken from queue 3 otherwise.

The code for threads 1 and 2 differs only in the accessed queues (queues 0 and 2 vs. 1 and 3). Furthermore, their code is nearly identical to the code for thread 2 in the pipelined case (Figure 5). Note however that the procedure call to *f* is now replaced with a call to *f_main*, a rewritten version of the procedure where the ordered sections have been replaced by code that stores a code fragment ID along with the read set of the ordered section in a queue. This queue is stored in the communication structure and is read by threat 3 that takes the ordered sections from the queue in original program order and executes them. This is modeled by the call to *f_ordered*. This procedure is independent of the control flow of *f*. It merely executes the ordered sections.

## 6.4 Rewriting Callee Procedures

The last step is to rewrite the procedures called from a parallelized loop. These procedures must be modified to utilize privatized copies of memory objects (if any). Also, if they contain ordered sections, then these must be factored out and their update and read sets must be queued up in the communication structure. Rewriting callees is a recursive process: the callees of callees must also be rewritten.

# 7. EVALUATION

We evaluate the Paralax compiler on the SPECint benchmarks bzip2 and mcf. For bzip2, KILL annotations were added in order to expose loop-level parallelism [24]. The parallelism in mcf was discovered autonomously by the Paralax compiler.

Execution times are measured on a 2.3 GHz AMD Opteron 2378 quad-core dual processor with Scientific Linux 5.3, kernel version 2.6.18. The LLVM revision number 83199 is used. The benchmarks are executed on SPEC reference inputs. For each measurement point, the benchmarks were run 10 times and the average execution time is reported. The single-threaded performance point corresponds to using the same compiler setup and optimization flags to generate single-threaded code (which does not include overheads due to parallelization).

Thanks to the annotations, the compiler finds two parallel loops in bzip2: the main compression loop and the main decompression loop. The compression loop is a parallel-stage pipeline that achieves a speedup of up to 2.34. The parallel stage corresponds to the actual parallelization while the sequential stages cover input and output of the data. Execution time starts to vary at 6 threads because of measurement noise and because there is insufficient parallelism to movitivate the use of this many threads.

The decompression loop is a highly unbalanced 2-stage pipeline, achieving the fundamentally limited speedup of 1.15. Overall, speedup is 1.79 on 8 threads. The graph also shows that without adding annotations, no exploitable parallelism is discovered. The annotations consist of annotating C library functions and inserting 14 KILL annotation statements in total.

The mcf benchmark has a loop that takes an important part of the execution time and can be parallelized without inserting annotations. The loop is parallelized as a 2-stage pipeline where the first stage is parallel. This structure yields a scalable speedup, up to a factor of 6.03 on 8 threads. Because this loop is responsible for only about 60% of the serial execution time, the overall speedup is limited to 2.06 (Amdahl's Law).

# 8. CONCLUSION

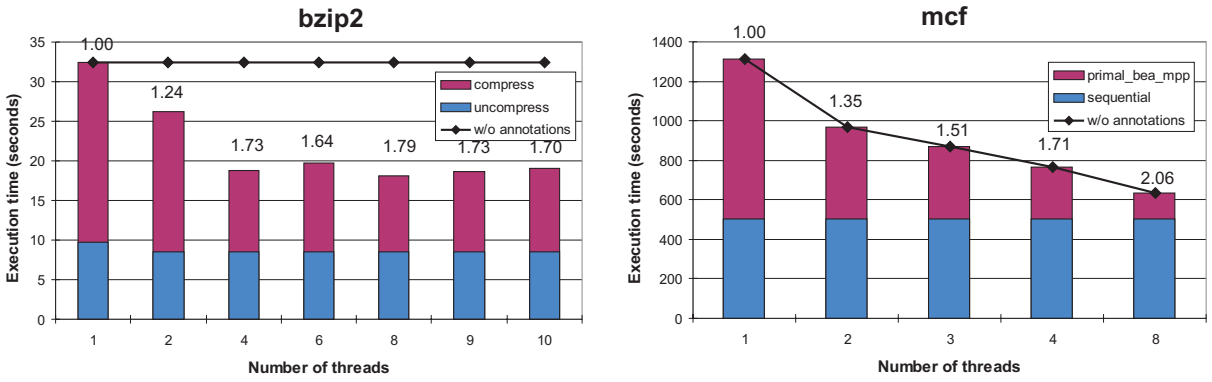
Implicitly parallel programming aims to improve the programmability, efficiency and correctness of parallel programs by automatically parallelizing sequential programs where crucial semantics have been specified as annotations in the source code. The genericity of the programs that we aim to parallelize however drives the design of the annotations and the automatic parallelizing compiler, due to the occurrence of system calls in parallel loops, pipeline-parallel loop structures, etc.

This paper describes the Paralax compiler and the annotations that it recognizes. We have discussed the design of the compiler, how it aims to parallelize our target programs and we have validated its design by applying it to the parallelization of SPECint programs.

Future extensions of Paralax are to support more annotations to sharpen the semantics of programs for particular constructs, such as assertions, potential reordering of I/O and other system calls, analyzing multi-dimensional matrices represented as arrays of pointers in C, etc. Furthermore, we intend to complement these annotations with algorithms that can identify suitable locations to insert them by using dynamic analysis information.

# Acknowledgments

Hans Vandierendonck is a Postdoctoral Fellow of the Research Foundation – Flanders (FWO). This work is supported in part by Ghent University, by the Institute for the Promotion of Scientific-Technological




(a) bzip2 (b) mcf

**Figure 8: Speedups obtained on the parallelized loops and total execution for the bzip2 and mcf benchmarks.**

Research in the Industry (IWT) and by the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC).

# 9. REFERENCES

[1] S. Amarasinghe. The looming software crisis due to the multicore menace, Feb. 2007. NSF Lecture.

[2] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *Proc. of 24th OOPSLA*, pages 97–116, 2009.

[3] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *Proc. of 40th MICRO*, pages 69–84, 2007.

[4] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *Proc. of '07 PLDI*, pages 223–234, 2007.

[5] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[6] W.-M. e. Hwu. Implicitly parallel programming models for thousand-core microprocessors. In *DAC-44*, 2007.

[7] J. C. Jenista, Y. h. Eom, and B. C. Demsky. Ooojava: software out-of-order execution. In *Proc. of 16th PPoPP*, pages 57–68, 2011.

[8] A. Kejariwal, A. V. Veidenbaum, A. Nicolau, M. Girkar, X. Tian, and H. Saito. On the exploitation of loop-level parallelism in embedded applications. *ACM Trans. Embed. Comput. Syst.*, 8(2):1–34, 2009.

[9] J. R. Larus. Loop-level parallelism in numeric and symbolic programs. *IEEE Transactions on Parallel and Distributed Systems*, 04(7):812–826, 1993.

[10] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proc. of '07 PLDI*, pages 278–289, 2007.

[11] G. Ottoni and D. I. August. Communication optimizations for global multi-threaded instruction scheduling. In *Proc. of 13th ASPLOS*, pages 222–232, 2008.

[12] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proc. of 38th MICRO*, pages 105–118, 2005.

[13] J. M. Perez, R. M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multicore architectures. In *Proc. of '08 CLUSTER*, pages 142–151, Sept. 2008.

[14] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multi-threaded transactions. In *Proc. of 15th ASPLOS*, pages 65–76, 2010.

[15] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *Proc. of 6th CGO*, pages 114–123, 2008.

[16] S. Rul, H. Vandierendonck, and K. De Bosschere. A profile-based tool for finding pipeline parallelism in sequential programs. *Parallel Computing*, 36(9):531–551, 2010.

[17] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *Proc. of 40th MICRO*, pages 356–369, 2007.

[18] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *Proc. of 41st MICRO*, pages 330–341, 2008.

[19] P. Tu and D. A. Padua. Automatic array privatization. In *Proc. of 6th LCPC*, pages 500–521, 1994.

[20] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *Proc. of 16th PACT*, pages 49–59, 2007.

[21] H. Vandierendonck and K. De Bosschere. Whole-array SSA: An intermediate representation of memory for trading-off precision against complexity. In *Proc. of Workshop on Intermediate Representation*, 8 pages, Apr. 2011.

[22] H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos. Parallel programming of general-purpose programs using task-based programming models. In *Proc. of 3rd HotPar*, 6 pages, May 2011.

[23] H. Vandierendonck, S. Rul, and K. De Bosschere. Factoring out ordered sections to expose thread-level parallelism. In *Proc. of '09 PESPMA*, page 8, June 2009.

[24] H. Vandierendonck, S. Rul, and K. De Bosschere. The Paralax infrastructure: automatic parallelization with a helping hand. In *Proc. of 19th PACT*, pages 389–400, 2010.

[25] H. Zhong, M. Mehrara, S. A. Lieberman, and S. A. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *Proc. of 14th HPCA*, pages 290–301, 2008.