Tetra: Evaluation of Serial Program Performance on Fine-Grain Parallel Processors *

Todd M. Austin
Gurindar S. Sohi

Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, WI 53706
{austin; sohi}@cs.wisc.edu

July 11, 1993

Abstract

Tetra is a tool for evaluating serial program performance under the resource and control constraints of fine-grain parallel processors. *Tetra*'s primary advantage to the architect is its ability to quickly generate performance metrics for yet to be designed architectures. All the user needs to specify is the capabilities of the architecture (e.g., number of functional units, issue model, etc.), rather than its implementation.

Tetra first extracts a canonical form of the program from a serial instruction trace. It then applies control and resource constraint scheduling to produce an execution graph. The control and resource constraint scheduling is directed by a processor model specification supplied to the program. Once scheduled, *Tetra* provides a number of ways to analyze the program's performance under the specified processor model. These include parallelism profiles, storage demand profiles, data sharing distributions, data lifetime analysis, and control distance (branch, loop, and call stack) distributions.

In this report, we present the extraction, scheduling, and analysis methodologies used by *Tetra*. We detail their implementation and discuss a number of performance optimizations used. The appendix includes the user's documentation for *Tetra*.

*© 1993 by Todd M. Austin and Gurindar S. Sohi. This work was supported by a grant from the National Science Foundation (grant CCR-8919635).

1 Introduction

A quantitative analysis of program execution is essential to the computer architecture design process. With the current trend in architecture of enhancing the performance of uniprocessors by exploiting fine-grain parallelism, first-order metrics of program execution, such as operation frequencies, are not sufficient; characterizing the exact nature of dependencies between operations is essential.

To date, most processors have either executed instructions sequentially, or have overlapped the execution of a few instructions from a sequential instruction stream (via pipelining). For such processors, the relevant characteristics of dynamic program execution included operation frequencies, branch prediction accuracies, latencies of memory operations in cache-based memory systems, amongst others. The continuing trend in processor architecture is to boost the performance (of a single processor) by overlapping the execution of more and more operations, using fine-grain parallel processing models such as multiscalar, VLIW, superscalar, decoupled, systolic, or dataflow (in a complete or a restricted fashion). To aid the design of such processors, simple first-order metrics of the dynamic execution, such as operation frequencies and operation latencies, are not sufficient by themselves. What is needed is a thorough understanding of how the operations of a program interact, *i.e.*, what is the nature of dependencies between operations in the dynamic execution graph, how these are impacted by the compiler and processor model, and how they impact performance. To this end, we employ dynamic dependence analysis.

Dynamic dependence analysis uses a canonical graph representation of program execution called the dynamic dependence graph (DDG). The nodes of this graph represent operations executed, and the edges in the graph represent dependencies between operations. To perform dynamic dependence analysis, we first extract the DDG from a serial program, and then re-schedule it based on the control and resource constraints of an abstract compiler and processor model. We then analyze the resulting execution graph to see the impact of the execution model on the program's performance. By iterating through this process, we can begin to more effectively explore the trade-offs involved in the design of processors that aggressively exploit fine-grain serial program parallelism.

The primary advantage of dynamic dependence analysis over other program evaluation techniques (e.g., function simulation, timing simulation, or prototyping) is its ability to quickly generate performance metrics for yet to be designed architectures. The actual implementation of the compiler or processor need not be known; all that must be specified to perform analysis is the *capabilities* of the model.

In this report, we present the extraction, scheduling, and analysis methodologies used in our dynamic dependence analyzer, *Tetra*. In Section 2 we introduce the notion of the dynamic dependence graph and discuss the various dependencies that exist in program executions. We show how DDGs can be extracted from serial traces, and then bound to a particular execution model via control and resource scheduling. A number of performance metrics that can be obtained from scheduled DDGs are described. In Section 3 we present algorithms for extracting, scheduling, and analyzing serial program DDGs. We also explore the time and storage complexities of the presented algorithms and previous work is cited. In Section 4, the implementation of *Tetra* is discussed and we detail a number of successful performance optimizations that were applied. Finally, we summarize the report in Section 5.

2 Overview

The primary structure that is constructed, manipulated, and analyzed is the *dynamic dependence graph (DDG)*. It is a partially ordered, directed, acyclic graph representing the execution of a program for a particular input. The executed operations comprise the nodes the graph and the dependencies realized during the execution form the edges of the graph.

While a particular operation in the DDG corresponds to an single instruction in the program's executable, we will not use the terms *operation* and *instruction* interchangeably. There are two reasons for this. First, because the DDG captures a complete execution of a program, there will be a one-to-many mapping between the instructions in the program's executable and the operations in the DDG. For example, repeated execution of a loop will see the loop instructions represented as many times in the DDG as the number of iterations executed in the program's execution. Thus, an operation is actually an instance of a particular instruction. Second, some instructions do not create values, and thus will not find themselves in the DDG. Examples of these instructions include unconditional branches, and NOPs.

These instructions are more so artifacts of the particular architecture the analyzed program was compiled for; we leave them out of the actual DDG.

The edges in the DDG force a specific order on the execution of dependent operations – forming the complete DDG into a weak ordering of the program's required operations. Some of these program dependencies are inherent to the execution of the program and cannot be removed without changing the algorithms used in the program, others can be removed, but usually not without costs in storage and possibly execution speed. We classify program dependencies into three categories: data, control, and resource.

Data Dependencies: Two operations share a data dependency if one operation creates a value that is used by the other (also called a Read-After-Write, RAW, or flow dependency). This dependency forces an order upon operations such that source values are created before they are used in subsequent operations. Data dependencies can only be removed by changing the algorithms used in the program and/or with compiler transformations that rewrite the code. Examples of algorithmic changes include the use of parallel algorithms (*e.g.,* parallel reduction) instead of sequential algorithms (*e.g.,* serial summation). Examples of compiler transformations include loop unrolling, invariant loop code motion, loop interchange, and strength reduction. A DDG which contains only data dependencies, and thus is not constrained by any control or resource limitations, is called a *dynamic data flow graph.* Under some execution models, extra ambiguous data dependencies may be inserted into the DDG. Ambiguous data dependencies occur when static *memory disambiguation*, performed in the compiler, cannot determine if a memory read and write access (*e.g.*, pointer dereferences) do or do not access the same memory, hence a conservative assumption must be made that they do, and an ordering must be enforced during compilation such that the write precedes the read instruction. In the actual execution, the same ordering will result unless the hardware has support for speculative loads (such as in the multiscalar processor [FS92]).

Control Dependencies: Control dependencies are introduced by conditional branches encountered during the execution of a program. Until the branch's predicate has been resolved, it is not known which instructions will be executed after the branch, thus all subsequent instructions share a control dependency with the conditional branch. This view of control dependence is actually over-restrictive, since it has been shown [FOW87] that by applying control dependence analysis, it is possible to either remove some control dependencies or re-attribute them to earlier executed instructions. If an instruction in the control flow graph post-dominates a conditional branch, that instruction need not share a control dependence with the conditional branch, since any path taken after the conditional branch, will always execute the post-dominating instruction. The instruction will instead be control dependent on an earlier executed predicate or with nothing (*i.e.*, the instruction's execution is invariant with respect to any decisions made in the program. We will utilize control dependence analysis in Section 3 to design less restrictive issue models. In either case, control dependencies can be removed via speculative execution. If a processor correctly guesses the outcome of a branch, execution can proceed as if the control dependence never existed. Of course, when using speculative execution, the effectiveness of a processor's ability to remove control dependencies is a function of the program, its execution, the branch prediction algorithm, and the processor's ability to "back out" of its mis-predicted paths.

Resource Dependencies: Resource dependencies (sometimes called structural hazards) occur when operations must delay because some required physical resource has become exhausted. Examples of limited processor resources include functional units and physical storage. It is always possible to remove resource dependencies by supplying more of the critical resource.

A very important resource dependency to consider is the storage dependency. Storage dependencies occur when parallel operations compete for a single physical storage location. Storage dependencies require that a computation be delayed until the storage location for the result is no longer required by any previous computation. These dependencies are often further classified into Write-After-Read dependencies (also called anti-dependencies or WAR hazards) and Write-After-Write dependencies (also called output-dependencies or WAW hazards). Since many different data values can reside in a single storage location over the lifetime of a program, synchronization is required to ensure a computation is accessing the correct value for that storage location. Violation of a storage dependency would result in the access of an uninitialized storage location, or another data value stored in the same storage location. Storage dependencies can always be removed by *renaming*. This technique (applicable in the hardware or the compiler) assigns a new physical storage location to each value created, making it possible for values destined for the same logical storage location to have overlapping lifetimes. If all values created are assigned to different locations, the program has the property of *single-assignment*; that is, a location is assigned to at most once. Of course this is wasteful to actually implement, as many values will have non-overlapping lifetimes, thus most renaming implementations will determine when a value is not longer needed thereby allowing its physical storage to be reclaimed.

2.1 Extraction

The DDG representation of program execution lends itself well to parallelism studies. It lacks the total order of execution found in the serial stream; all that remains is the weakest partial order that will successfully perform the computation required by the algorithms used. If a machine were constructed to optimally execute the DDG, its performance would represent an upper bound on the performance attainable for the program. This claim can be made largely independent of the compilation strategy of the program, since the DDG does not contain control or resource dependencies. While the compiler can change some data dependencies, these changes are typically local in nature. [1] To make any substantial impact on the form of the DDG would require changing the algorithms used in the program. Since compilers cannot generally perform these changes, we say that the DDG approaches a canonical form of a program. Later, we will re-map this canonical form to a processor model that we wish to study, and reveal its impact on the program's performance.[2]

Figure 1 shows a simple program fragment, its execution trace, and the resulting DDG. Note how the loop body instructions in the original program create many instances of operations in the trace and DDG. This DDG only contains data dependencies (shown as solid arrows). In fact, we will only represent the data dependencies in the DDG; the control and resource dependencies will be accounted for when we bind the DDG to a processor model, via control and resource scheduling.

2.2 Scheduling

The process of scheduling operations binds them to a *level* in the execution graph. This binding is directed by an execution model specification. In the DDG of Figure 1(c) there are four levels, each one is a horizontal slice of the DDG. The scheduled DDG can be thought of as sequence of parallel instructions that would issue level-by-level in an abstract processor. In Figure 1(c), the DDG is scheduled for a processor model with no control or resource limitations, *i.e.,* an operation can execute as soon as its inputs are available. For this "oracle" processor, operations `I1` and `I2` would execute in the first cycle, `I3`, `I4`, `I6`, and `I11` in the second cycle, and so on.

There are two phases to scheduling a DDG: first, control scheduling, then resource scheduling.

In the first phase, *control scheduling*, we determine at what level the operation is first issued. The resulting schedule depends on the compiler and processor issue model as well as the control structure of the program. Two control scheduling models are explored in this work, *control flow graph (CFG) scheduling*, and *control dependence graph (CDG) scheduling*. The first model, CFG scheduling, is the traditional issue model where instruction issue is based on control dependencies realized on a traversal of the control flow graph of the program. The second model, CDG scheduling, uses the less restrictive control dependence graph [FOW87]. Both scheduling algorithms are described in Section 3.4.

In the second phase of scheduling, *resource scheduling*, the issued operations are allocated to processor resources using a scheduling heuristic. Two processor resources must be allocated before an instruction can be executed – functional unit and storage resources. Functional units are allocated just long enough

[1] Some program level transformations, such as recurrence breaking and induction variable expansion, can have a major impact on the data dependencies of loop intensive programs. Still these changes are always quite regular, and can be easily simulated during DDG extraction – algorithmic changes, for instance, identifying and transforming a serial FFT to parallel FFT, are not yet attempted by compilers and will likely never have widespread applicability.

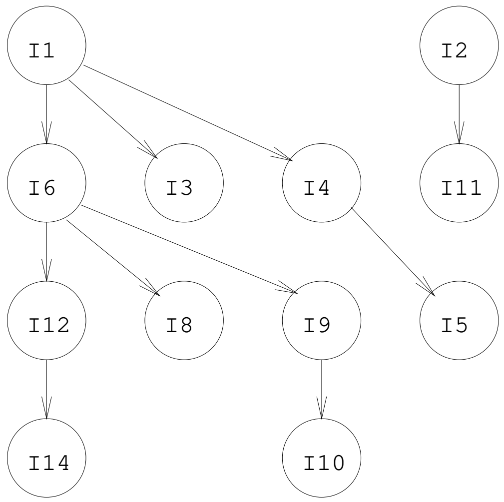
[2] In this respect, the compiler is very relevant, because it has significant control over how control and resource dependencies will manifest. Hence we want to remind the reader that an execution model includes both the hardware and the compiler techniques applied.

```
        r1 <- 2
        r3 <- 0
loop:   beq r1,done
        r2 <- r1 % 2
        beq r2,next
        r3 <- r3 + 1
next:   r1 <- r1 - 1
        b loop
done:
```

a)

```
I1:  r1 <- 2
I2:  r3 <- 0
I3:  beq r1,done
I4:  r2 <- r1 % 2
I5:  beq r2,next
I6:  r1 <- r1 - 1
I7:  b loop
I8:  beq r1,done
I9:  r2 <- r1 % 2
I10: beq r2,next
I11: r3 <- r3 + 1
I12: r1 <- r1 - 1
I13: b loop
I14: beq r1,done
```

b)



c)

Figure 1: Sample program *a)*, execution trace *b)*, and resulting DDG *c)*. The solid lines in figure *c* represent true data dependencies, and the node labels correspond to an instruction instance in figure *b)*. **beq <reg>,<label>** branches to **<label>** if the value in register **<reg>** is zero, otherwise, it falls through to the next instruction. The "%" operator computes the remainder of an integer divide.

to compute the result value which is then stored in the allocated physical storage (either register or memory). The physical storage is reserved for an indeterminate amount of time. We could extend the resource scheduling to include other processor resources, such as result buses, functional unit input buses, or instruction window slots (as in [AS92]); the methodologies described here would apply to these cases as well.

We study both unconstrained and constrained resource scheduling techniques – all of which are detailed in Section 3. The first, *topological sorting*, is an optimal, resource unconstrained scheduling technique. We define an *optimal schedule* to be one in which the given supply of resources produces the shortest length schedule. Unfortunately, generating an optimal schedule of a DAG in the presence of resource limitations is an NP-complete problem for even very small resource supplies [LK78, GGJ78, GJ79]. Thus no known algorithm can do the task is less than exponential time (exponential with respect to the size of the trace being analyzed!) When resources are limited, a scheduling heuristic must be applied and the resulting schedule will nearly always be sub-optimal.

We propose five resource constrained scheduling heuristics: history schedule (HISTORY), list schedule–best fit (LIST-BF), list schedule–first fit (LIST-FF), round robin (RND-RBN), and random (RANDOM). The heuristics were selected for study because they exhibit varying degrees of storage and space complexity during execution, and in return, provide equally varying degrees of performance with respect to an optimal schedule. In Section 3.5 we describe each of the heuristics as well as show their relative performance with respect to each other. Once scheduled, we can derive a number of interesting metrics from the execution graph.

2.3 Analysis

DDG metrics fall into two categories: resource demand profiles and value metrics. The first, *resource demand profiles*, show the quantitative demand for a particular resource as a function of the level in the scheduled DDG. The *parallelism profile* is a resource demand profile showing the number of functional units used in each level of the DDG. Figure 2 shows the parallelism profile of the GNU "gcc" compiler compiling its own source file "cexp.c".

The *available parallelism* is defined as the arithmetic average number of operations per level in the parallelism profile and can be viewed as the speedup that could be attained by an abstract machine capable of extracting and executing the DDG from the program. The length of the parallelism profile is the *critical path length* of the execution. This length is identical to the height of the scheduled DDG, and it gives the minimum number of steps required to evaluate the operations in the scheduled DDG. In Figure 1(c), the available parallelism is $\frac{2+4+4+2}{4} = 3$, and the critical path length is 4. In Figure 2 the available parallelism is 42 operations/level and the critical path length is approximately 550,000 levels.

Values used by parallel operations must reside in physical storage. The *physical storage demand profile* (or waiting token profile) shows the amount of storage required at each level of the DDG. This profile is a function of both the parallelism in the program, and the lifetime of the values: as each increases, so will the physical storage requirements. This metric is useful in determining the amount, and type, of temporary storage required to realize the levels of parallelism shown in the parallelism profile. In Figure 3, the aggregate, compacted storage demand profile for GCC is shown. *Tetra* can also break the storage down by register and memory storage demand. It can do this either by the storage class of the original reference (*i.e.*, register or memory), or by the *likelihood* that a compiler would allocate the object to a register or memory location. In the latter case, register locations are reserved for high frequency and short lived values, while memory is reserved for all other values. This method recognizes that storage allocation in a compiler is inextricably bound to the architecture upon which the code is intended to run, thus the further the host architecture departs from the abstract architecture, the less likely register allocation will be the same. Using the frequency/lifetime based breakdown, it becomes possible for the architecture designer to get a handle on how much storage will be needed in each level of the memory hierarchy to achieve a given level of program performance.

Value metrics examine characteristics which reveal of how values are produced, manipulated, and consumed during a program's execution. For example, the *value lifetime distribution* shows the distribution of value extents. The extent of a value is measured as a distance with respect to levels of the DDG. Figure 4 shows the value lifetime distribution of the sample GCC execution.

The *degree of sharing distribution* is a frequency distribution indicating how many other nodes in the

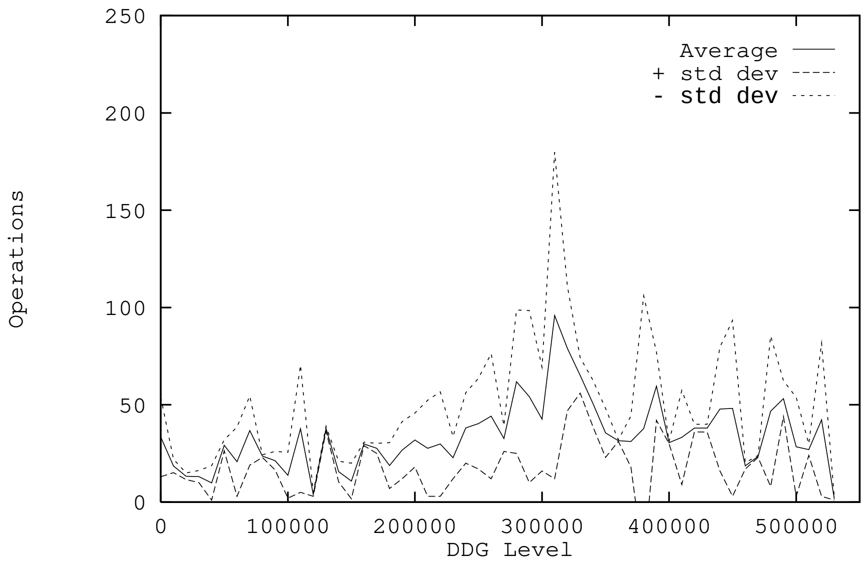


Figure 2: Example Parallelism Profile. GNU “gcc” compiling “cexp.c”. This parallelism profile has been compacted; the top dashed line is the average plus the standard deviation, the bottom dashed line is the average minus the standard deviation, and the middle solid line is the average parallelism over 10,000 levels.

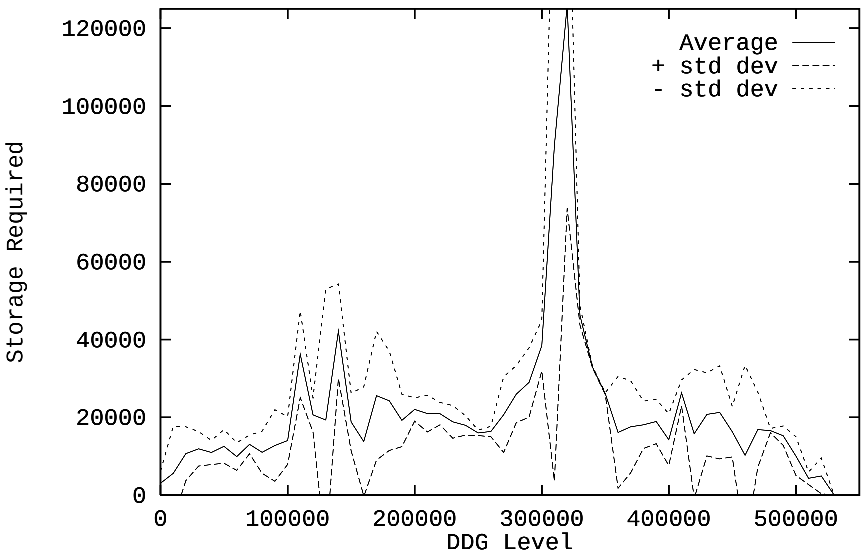


Figure 3: Example Aggregate Storage Demand Profile. GNU “gcc” compiling “cexp.c”. This parallelism profile has been compacted; the top dashed line is the average plus the standard deviation, the bottom dashed line is the average minus the standard deviation, and the middle solid line is the average parallelism over 10,000 levels.

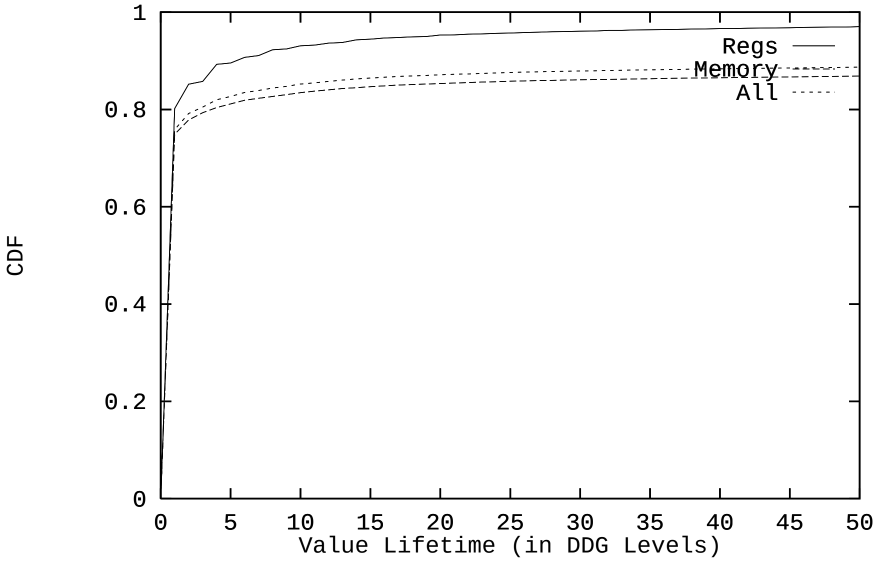


Figure 4: Example Value Lifetime Distribution. GNU "gcc" compiling "cexp.c". This graph is a cumulative distribution function showing the lifetime, in DDG levels, of register and memory values (as allocated by the host compiler).

DDG used generated values. Such a distribution is important not only for a possible dataflow realization of our abstract execution engine (by indicating how many operations can be "fired" when a token is created in an explicit token store machine, for example), but also for more conventional multiprocessor execution of the program. In a multiprocessor, different processors are responsible for execution of different parts of a (suitably constrained) DDG of a program. By measuring how much data flows from the nodes in one subgraph to another (albeit in a very constrained form of the DDG that matches the processor execution model), we can measure the degree of data sharing amongst the processors, for example. These results would be directly applicable to the design of directory based cache coherency schemes. Figure 5 shows the degree of sharing distribution for the sample GCC execution.

Dependency distance distributions show the "control distance" between the producer of a value and a consumer. By examining this, it is possible to gauge how well a real architecture would have to speculate past program control structures to realize a given level of parallelism. Control distances can be measures in terms of any program control construct. In this work, we measure distances on the control flow graph, *branch distance*, distances between iterations of a loop, *loop distance*, and distances on the execution call stack, *call stack distance*. Figure 6 shows the branch, loop, and call distance for the sample run of GCC.

The branch distance reveals the level of branch speculation required by a simple, single window of execution processor to achieve the same level of performance as the abstract target model. The loop distance metric displays the balance between loop parallelism and functional (non-loop) parallelism. This is a good indicator of how well the abstract processing model will have to support either type of parallelism, either in the compiler or the processor implementation. And the call distance indicates how parallelism is spread across separate function bodies, and thus reveals how effectively the abstract architecture will have to either support dynamic execution of multiple procedures or static inlining.

Having introduced the concept of the dynamic dependence graph and described the process of DDG analysis, namely, extraction, scheduling, and analysis – we now dive head long into the details of how these processes can be implemented.

3 DDG Extraction, Scheduling, and Analysis Methodologies

Designing an algorithm to extract, schedule, and analyze a serial trace, of virtually unlimited length, is a challenging task. An interesting, representative trace can easily become larger than what could be

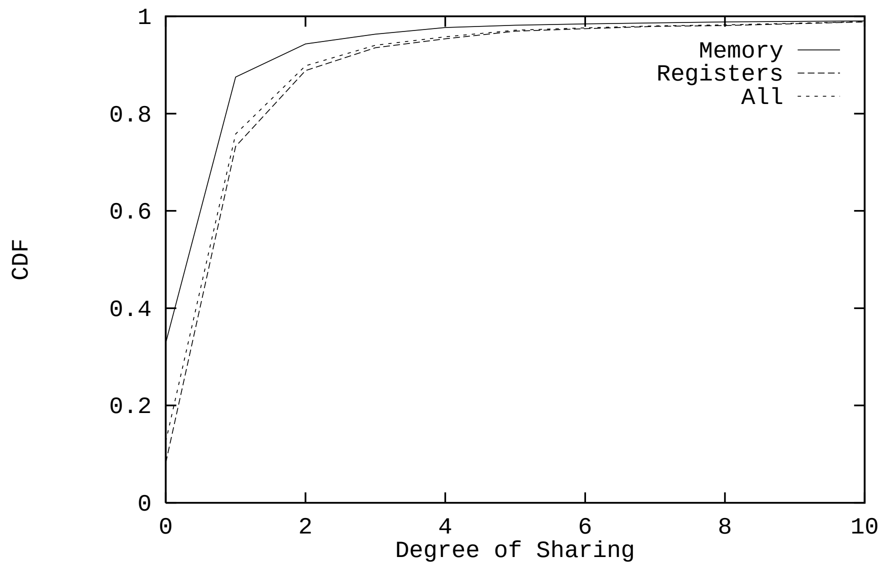


Figure 5: Example Degree of Sharing Distribution. GNU "gcc" compiling "cexp.c". This graph is a cumulative probability distribution showing the total sharing of values by register and memory values (as allocated by the host compiler).

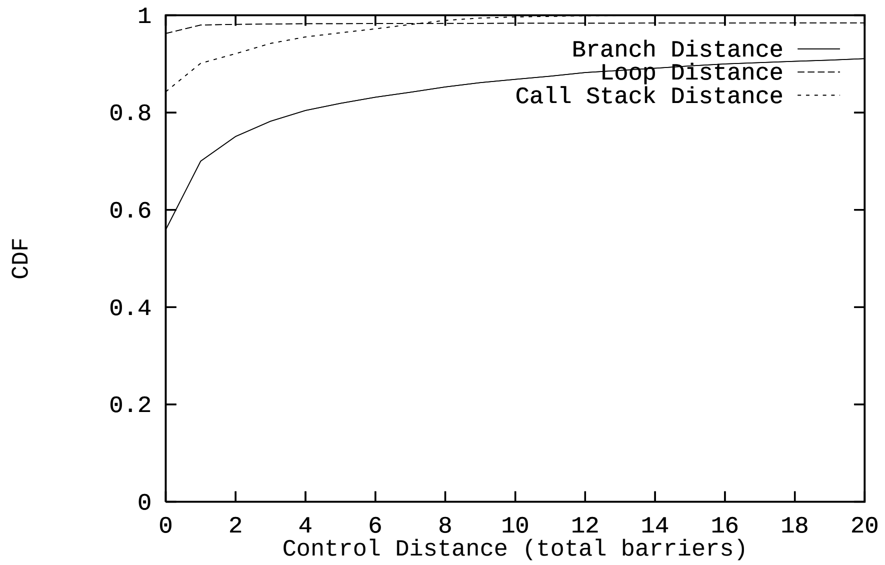


Figure 6: Example Control Distance Distribution. GNU "gcc" compiling "cexp.c". This graph is a cumulative probability distribution showing the total branch, loop, and call distance distributions of all edges in the DDG.

stored in a conventional computer memory. The resulting DDG, if annotated for analysis, can become even larger. The algorithms presented in this section have one dominating goal: limit storage usage at all costs, even if the time complexity of the algorithm must be increased. Before we describe our methodology, we briefly discuss the previous work in this area.

3.1 Previous Work

There has been a plethora of work which measured the average parallelism in a (sequential) instruction stream for a particular hardware configuration [BYP$^+$91, NF84, SJH89, TF70, Wal91]. These studies typically find the length of the critical path through the computation, and compute the average parallelism as the total number of instructions divided by the number of cycles of execution. (Some also measure the resource load to determine the maximum number of resources required). These studies typically evaluate how the average parallelism changes under various constraints such as register renaming, various branch prediction strategies, memory disambiguation strategies, changes in operation latencies, instruction window sizes, resource constraints, etc. Changes in a parameter result in changes to the critical path length (and resource demand), and consequently to the average (and maximum) parallelism. Because they are interested in only a single measure, namely the average or available parallelism, and not in other aspects of the DDG, they do not need to construct the entire DDG, or even parts of it.

An early work measuring total available parallelism is that of Kuck, et al.[Kea74] They statically analyzed (by hand) program dependencies in FORTRAN programs, and the resulting available parallelism was estimated from the analyzed code fragments. Kumar [Kum88] presented the first work that gathered exact parallelism profiles in serial FORTRAN programs. Kumar extracted parallelism profiles by rewriting FORTRAN programs (with COMET) such that the profile was generated during the execution of the program. Although the method would lend itself to any imperative language, it was only applied to FORTRAN. The operations in the DDG were FORTRAN statements which were assumed to execute in one unit of time. Our method of extracting DDGs from serial traces is quite different than Kumar's. While Kumar placed FORTRAN statements into the DDG, we instead place machine instructions into the DDG. This allows more precise control over the relative time taken for operations in the DDG, and finer-grain parallelism (such as that found *within* FORTRAN statements) will show up in our results. Moreover, since our technique builds the DDG from a serial execution trace, without modifications to the source program, it can be applied to any language, compiled or interpreted. Kumar implemented a control model similar to our CDG issue model. Specifically, he did not allow any FORTRAN statement to execute until all surrounding predicates had completed evaluation. Lam and Wilson's work [LW92] has shown this assumption to be overly restrictive, as the use of speculation can allow many operations to execute prior to the resolution of their surrounding predicates. While we too look at the effects of the control model on available parallelism, we also look at control barrier from another viewpoint. By measuring the control distance of dependencies, we can then show for a given level of parallelism, how effective speculation must be to expose that parallelism.

MaxPar [CSY90] is very similar to Kumar's COMET in that it also rewrites programs. Where Kumar's COMET was limited to scheduling program statements, MaxPar has the ability to schedule at the granularity of operation-level, statement-level, loop-level, or subprogram-level. It can also limit computing resources available during analysis. They implemented four list scheduling algorithms: near-optimal (identical to LIST-BF), earliest available (identical to LIST-FF), circular (identical to RND-RBN), and random (identical to RANDOM). We expand on their work by proposing a new, more powerful scheduling heuristic, HISTORY. We also extend their heuristics to support scheduling of resource with non-deterministic latency, which allows us to examine the effects of limited storage resources.

A number of papers from the dataflow literature have included examples of DDG analysis [CA88, ACM88, AN89, Nik90]. For example, Culler and Arvind [CA88] provide detailed parallelism profiles and waiting token (or storage demand) profiles for some dataflow programs. Their dataflow processor and language environment lends itself well to DDG analysis. Instrumenting the dataflow processor's execution is sufficient to generate the parallelism profiles and critical path. Yet their environment lacks storage and most control dependencies, so their results will not include the effects they have on available parallelism. Therefore, it is not clear how their results would extend to programs written in imperative languages such as C or FORTRAN, and also how they could be applied to processors that have more restricted computation models. We have found in our research that our results agree with the conclusions of [ACM88] and [Kum88] that ordinary programs and algorithms, not intended to execute in parallel environments, do indeed have a significant amount of fine-grain parallelism.

Larus performed detailed studies of loop level parallelism for a number of the SPEC benchmarks [Lar91]. Because the analysis was intended primarily for directing the development and application of compilers capable of parallelizing loops, the analysis was limited to intra-loop parallelism of each lexically top level loop. Our analysis examines parallelism in a more global view, thus allowing inter-loop parallelism to also be analyzed. Still, for a number of programs, our results are quantitatively very close to Larus', supporting that for many programs, much of the parallelism is intra-loop parallelism.

Mahlke *et al.* [MWC$^+$91] examined the effects of compiler optimizations on serial program parallelism. They looked specifically at three classes of program optimization: classical, superscalar, and multiprocessor. The superscalar optimizations encompass their superblock global trace scheduling techniques, loop unrolling and peeling, and induction variable expansion. The multiprocessor compiler optimizations were primarily memory renaming (simulated in the analysis) and data migration (faster loads). They found that classical and superscalar optimizations did expose more parallelism, especially in small instruction windows. And the multiprocessor optimizations were most effective for very large instruction windows. When running on an "oracle" machine (*i.e.*, no resource or control constraints, perfect memory disambiguation, and unlimited renaming), they found that program transformations exposed little more intrinsic parallelism. This supports our thesis that the DDG of a program is, for the most part, a canonical form of a program.

Wilson and Lam [LW92] studied the effects of control dependencies on available parallelism. Their primary result showed how combining control dependence analysis, multiple flows of control, and speculative execution could expose a large portion of the total available parallelism found in the unconstrained DDG. Hence, the critical factor in the development of aggressive fine-grain parallel processors will likely not be control dependence resolution, but rather other factors, like data dependence resolution. We extend their work by providing a framework in which both control and resource constraints can be combined.

Theobald, et al. [TGH92] examined the smoothability of serial program parallelism. By constraining the available computational resources (via a HISTORY schedule), they were able explore a program's sensitivity to limited functional unit supplies. We extend their work by evaluating the advantages of more computationally frugal scheduling heuristics as well as showing the limitations in their scheduling heuristic. We also examine the smoothability of the storage demand, and combine both control and resource scheduling.

We extend our earlier work [AS92] in a number of ways. This paper provides a more detailed description of the methodology, and we describe a number of invariants on the algorithm as well as examine its time and storage complexity. We extend our methodology to include construction of control and resource constrained DDGs, and then provide a full treatment of the scheduling heuristics, their complexity, and their limitations. We detail a number of very successful performance optimizations in our implementation of *Tetra*. And our results now include a number of new metrics, including storage demand profiles, value lifetime profiles, and control distance profiles.

3.2 Unconstrained DDG Extraction and Analysis

Construction of the unconstrained DDG need only consider the data dependencies realized in the analyzed program's execution. During scheduling, all control dependencies are ignored, and an unlimited supply of functional units and storage are available in any level of the DDG. Later, we will extend the unconstrained DDG methodology to also account for more restrictive issue models and limited physical resources, and we will discuss the many difficulties in their handling. The basic extraction, scheduling, and analysis algorithm is shown in pseudo-code in Figure 7.

The algorithm iterates across each instruction in the trace stream, in the order of their execution, performing extraction, control and resource scheduling, and analysis in one pass.

The trace instructions are accessed through the argument `trace`. This variable is a typed stream, and it functions similar to C's standard input stream. The trace stream provides two built in operations: EOF, and NEXT. `trace.EOF` returns a boolean indicating if there is more information to be read from the trace stream, and `inst := trace.NEXT` assigns the next instruction from the trace stream into the variable `inst`. Because the extraction and analysis is performed in one pass of the trace stream, the trace stream can be "piped" directly into the DDG analyzer, thus there is no need to store the entire instruction trace.

```
type
    Level: integer;
    UniqueID: Address OR RegisterNumber;
    Inst: record
            pc: Address;
            src: set of UniqueID;
            dest: UniqueID;
        end;
var
    baseLevel, deepestLevel: Level;

procedure AnalyzeDDG(trace: STREAM of Inst)
begin
    liveWell: MAPPING from UniqueID to Level;
    maxPred, value, schedLevel: Level;
    inst: Inst;

    baseLevel := 0;
    deepestLevel := 0;
    while (NOT trace.EOF) do
        inst := trace.NEXT;
        if (liveWell[inst.dest].EXISTS) then
            UpdateValueMetrics(liveWell[inst.dest]);
        endif
        if (HasSideEffect(inst)) then
            liveWell[inst.dest] := deepestLevel+Latency(inst);
            baseLevel := schedLevel := deepestLevel;
            deepestLevel := deepestLevel+Latency(inst);
        else
            maxPred := baseLevel;
            for all value ∈ inst.src do
                if (liveWell[value].EXISTS) then
                    maxPred := MAX(maxPred, liveWell[value]);
                endif
            endfor
            schedLevel := ControlSchedule(inst, maxPred);
            schedLevel := ResourceSchedule(inst, schedLevel);
            liveWell[inst.dest] := schedLevel + Latency(inst);
            deepestLevel := MAX(deepestLevel,schedLevel+Latency(inst));
            UpdateBarriers(inst, schedLevel);
        endif
        UpdateDemandProfiles(inst, schedLevel);
    endwhile
    for all value ∈ liveWell do
        UpdateValueMetrics(value);
    endfor
endproc
```

Figure 7: Main Loop of DDG Extraction, Scheduling, and Analysis Algorithm.

```
function ControlSchedule(inst: Inst, earliest: Level): Level;
begin
    ControlSchedule := earliest;
endfunc

procedure UpdateBarriers(inst: Inst, schedLevel: Level);
endproc

function ResourceSchedule(inst: Inst, earliest: Level): Level;
begin
    ResourceSchedule := earliest;
endfunc
```

Figure 8: Unconstrained DDG Control and Resource Scheduling Routines.

An operation's placement in the execution graph is first constrained by its input values, that is, it cannot be placed any earlier than its latest input value. This level is computed in the innermost `for` loop. The loop iterates across all the inputs to the current instruction, storing the most restrictive input into the variable `maxPred`. The initial value of `maxPred`, *i.e.,* the value of the variable `baseLevel`, forces a lower bound on the placement of operations into the DDG. `baseLevel` is used primarily in the handling of instructions with unknown side-effect, and in the handling of pre-existing values.

If an instruction has unknown side-effect, *e.g.,* system calls, it is not possible (or directly obvious) to identify what storage was read or written by the instruction. Therefore, the algorithm must either assume that the instruction modified all live values in the program (*pessimistic system call assumption*), or that it modified nothing (*optimistic system call assumption*). The conservative assumption is implemented by placing the operation after all levels currently used in the DDG and bounding the placement of future operations to levels later than the operation with unknown side-effect. This placement level, found in the variable `deepestLevel`, ensures that the operation has access to any value created previously. `deepestLevel` is updated every time a new level is used that is later than any previous level. The bounding of future operations is implemented by reseting the value of the variable `baseLevel`. Since all future placements can be no earlier than `baseLevel`, future instructions are forced to read any values updated by the instruction with unknown side-effect. Later, we will use an identical procedure to realize control dependencies. For optimistic system call assumptions, we simply place the instruction into the topologically earliest level in the DDG, (*i.e.,* at `baseLevel`).

Pre-existing values reside in the data segment. They are placed there at compile time by the compiler. On some architectures, these pre-existing values can also exist on the stack and in registers at the start of program execution. In any case, the value is placed on first reference, without a creating operation, into the execution graph such that it is available at the earliest defined level of the DDG. This level is stored in the variable `baseLevel`.

After determining the most restrictive input level (either that of the latest input, or the last system call), the calls to `ControlSchedule()` and `ResourceSchedule()` further constrain the placement to include the effects of control and resource dependencies, respectively. Control scheduling is always performed before resource scheduling, since an operation must first be issued before its resource requirements can be determined. `UpdateBarriers()` is later called to update any control barriers that might have been created by the current instruction. These routines are processor model dependent. We detail the ones implemented by *Tetra* in Section 3.4.

Figure 8 shows the scheduling routines used to schedule a DDG without resource or control constraints. Since there are no control or resource constraints, the routines simply return the passed levels. The resulting execution graph is the topologically sorted [Tar83] DDG. The schedule is optimal with respect to length, and it represents an upper bound on the performance that can be attained for the analyzed program. This schedule is also commonly called the eager evaluation schedule, because an operation is scheduled as soon as it its inputs are ready. This is only one of many possible optimal schedules for the operations; for example, the schedule in Figure 1(c) would still be an optimal schedule if the operation `I8` was moved down one level in the schedule.

Once an operation's level in the execution graph is known, we can compute when its result value is available as simply the operation's placement level plus the latency to create the value. The algorithm calls the processor dependent function `Latency()` to determine the latency of an operation. This latency is equal to the total time required to compute a result, or the total time that the resource must be reserved (as in the case of memory).[3] The level of the newly created value is then inserted into the *live well* under the address (or register specifier) of the destination storage.

The live well acts as an associative memory, allowing the algorithm to find the DDG level of an operation's input values, given a unique value identifier. The algorithm employs a mapping variable instead of an array, since the domain of value identifiers is likely to be large and sparse. The mapping variable, `liveWell`, supports the built in function `map[indexor].EXISTS`, which returns a boolean value indicating if there currently exists a mapping from `indexor` to some value. The assignment `map[indexor] := value` installs a new value into the mapping variable, deleting the previous value if one exists. In *Tetra*, the live well is implemented with a hash table.

It is not necessary to assign arbitrary (unique) value identifiers to newly created values. Because values residing in the same storage location will have non-overlapping lifetimes, it is sufficient to use the value's destination register number or memory address as a unique value identifier. The serial semantics of the trace stream will ensure that any reference to an entry in the live well always yields the correct value information. In the presented algorithm, we've assumed that a register number is obviously different than a memory address, if this is not the case, a single bit can be added to distinguish the two, or the registers can be mapped into an unused portion of the memory address space.

The storage requirements of the live well are proportional to the size of the program's *active name space*. The active name space of a program is the complete set of names (memory addresses and register specifiers) that a program will produce during its execution. Since we cannot ever de-allocate a value record, any touched memory will incur a memory cost. It is reasonable to assume that, if the value records are kept reasonable small, we can analyze nearly any program, since by definition the program's active name space must fit into the memory of the computer that can run the program. We reduce this overhead by trimming the per-value memory requirements in the live well, as well as by reducing the live well overhead. In Section 4 we discuss the implementation of the live well for *Tetra*. Since the live well is implemented with a hash table, the time complexity for accessing values is $O(1)$.

Generation of the resource demand profiles occurs at the call to `UpdateDemandProfiles()`. At this point, an operation has just been placed into the DDG. By inspecting the instruction, we can determine what resources are used, and update the appropriate profile vector at an index derived from the level of the placed operation. Because the profile's storage size will be on the order of the constructed execution graph's height (critical path length), it may be necessary to increase the granularity of the profile array entries. In other words, instead of saving the resource demand for a single level in an array element, we will save the total resource demand over n levels in an array element (producing a profile of granularity n). Unfortunately, no information other than the average parallelism (or storage demand) can be computed for the range. This is because the samples in the range, *i.e.,* the number of operations placed at each level, are not known until the end of the analysis. Other interesting information, such as the exact shape of the curve, or the maximum, minimum, or variance of the parallelism (or storage demand) is lost. This is usually only a problem for long traces with very little parallelism. We describe, in Section 4, a number of implementation tricks that can help mitigate the profile storage requirements.

Value metric distributions require information be stored with values in the live well. Whenever a value is created or referenced, those variables may have to be updated. For example, when computing the degree of sharing distribution, we keep a counter, initially zero, with the value in the live well, and increment it each time the variable is referenced. When the value becomes dead (at the call to `UpdateValueMetrics()`), the stored value usage information can then be transferred to the appropriate frequency distribution. It is important to also call `UpdateValueMetrics()` on all values in the live well upon completion of trace analysis, since all remaining values in the live well have become dead.

The actual mechanics to produce the value metric distributions is generally trivial, so we shall not overextend our welcome with the reader to describe them all here. However, since the implementation of the control distance metrics is significantly more complicated, we offer a complete description of these

[3] Later, we will also define a `PLatency()`, which is the amount of time that the resource must be reserved before another operation can use it – this is useful for building schedules for pipelined resources.

metrics and fully describe their handling in the following section.

3.3 Control Distance Analysis

Consider, for sake of illustration, we accepted the task of constructing a machine that could exploit all the available parallelism from a serial program's instruction stream. One major obstacle to accomplishing this task would be devising a mechanism that could effectively overcome the control barriers found between operations producing values, and operations consuming those values. For many values, the control barriers between the producer and consumer will have to be overcome in as little as one cycle of this abstract machine's execution.

Control barriers are program constructs that perturb the normal inline fetch and issue of instructions. We quantify the "difficulty of overcoming" control barriers as the number of barriers that must be resolved (or speculated) between the operation that produces a value, and the operation that consumed that value. Since this path is always represented by an edge in the DDG, we say that a DDG edges has a *control distance* with respect to the serial trace. By building control distance distributions, we can gauge just how effectively this "abstract" architecture will have to overcome control barriers before it can fully exploit the available parallelism.

We examine three control distances: branch distance, loop distance, and call distance. The *branch distance* of a DDG edge is defined as the number of conditional branches that must be resolved (or speculated) in the serial trace before the consuming operation can be issued. The *loop distance* is the number of loop iterations that must be issued before the consuming operations is issued. This is commonly referred to in the literature as a loop carried dependency distance. The *call distance* is the distance, on the call stack, between the producing operation and the consuming operation.

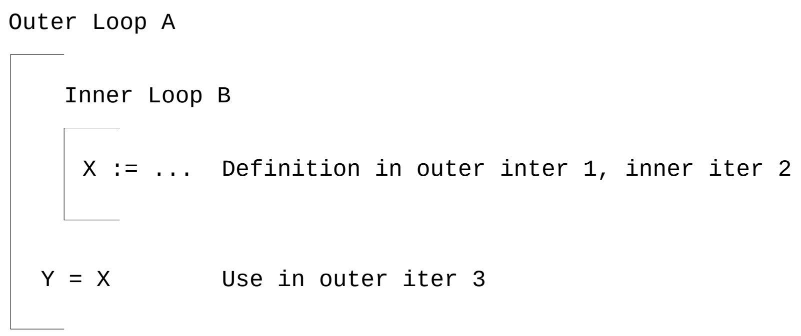
We need to examine two issues: first, which edges in the DDG are most meaningful to examine, and second, what procedure can we use to measure each type of control distance.

While we can always examine the control distances on all edges, we can gain clearer results by concentrating on only the edges whose control barriers will be the "hardest" to overcome. These edges are the ones on the critical path. Since a value traveling along a critical path edge must be transmitted to the consumer in one "cycle" of our abstract machine, these edges become the most important, and hardest to resolve. Unfortunately, its computationally infeasible to locate the exact set of edges on the critical path – instead we can use a property of the critical path to easily generate a reasonable superset of these edges. All edges on the critical path span exactly one level of the DDG. Thus if we analyze only edges of length one, we can trim away many of the "easier" edges from our metrics. Just how many of these "easy" edges are removed is a function of the program and its execution. We call the length one edges the *most restrictive edges*.

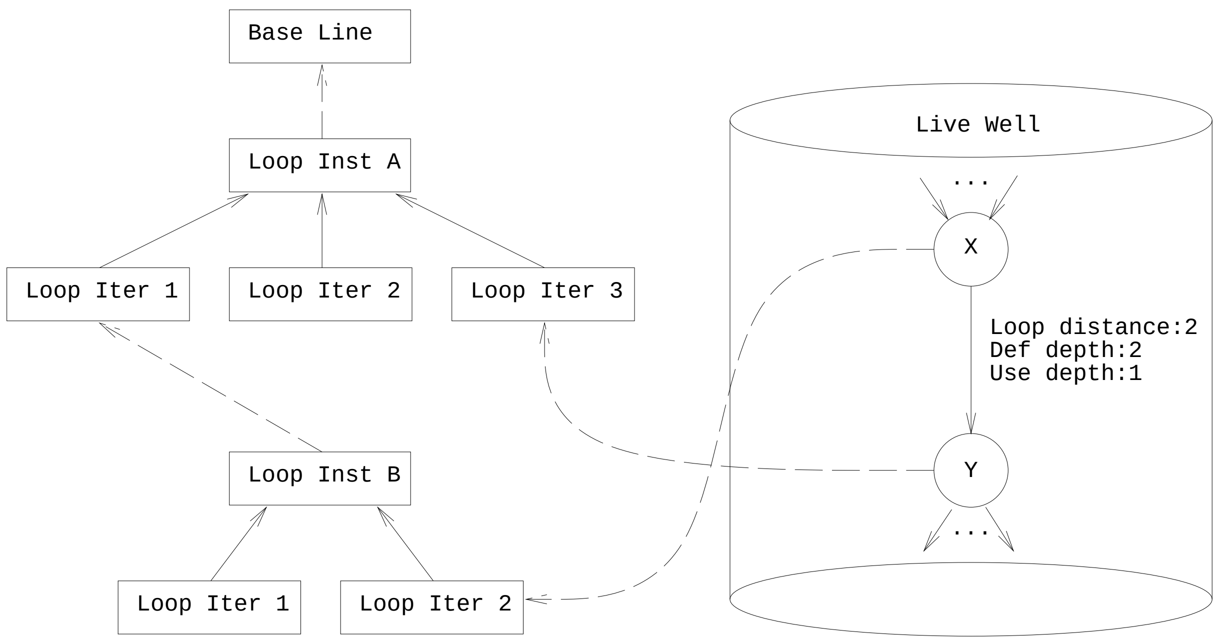
The simplest control distance to measure is the branch distance. The only state required is a global branch counter. This counter is incremented each time a conditional branch instruction is encountered. The branch counter's current value is then placed into the live well when a value is created. The branch distribution is updated each time another operation uses this value (or each time a length one edge is created from this value, if we are only examining the most restrictive edges). The distance is computed as the current branch counter value, minus the branch counter value stored in the value record.

The loop distance of an edge is defined to be the intra-loop iteration distance – it is zero if the definition of a value is not used in the same loop instance. An instance of a loop is one complete invocation of a loop structure in the program's execution. Only loops executed from within another loop may have multiple instances. Figure 9(a) shows a nested loop example.

In the example code of Figure 9, a definition of X is created in the first iteration of loop A while it is executing the second iteration of the nested loop B. This value, in X, is later used in the third iteration of loop A, outside of the scope of loop B. The loop distance, in this example, is on the edge between the operation that creates X and the operation that uses X (as shown in the live well of Figure 9(b)). The loop distance of this edge is two, since the definition of X occurs in the first iteration of loop A, and is used in the third iteration. The instrumentation methodology is complicated by the possibility that a value may be created in a dynamically nested loop, which may complete execution, and thus require that its values be attributed to the surrounding loop. In fact, we will not know which loop to attribute the value to until it is used – the first common loop in the tree of dynamic loop instances is the loop in which the value is defined and used. The intra-loop distance is then computed by subtracting the use iteration,



a)



b)

Figure 9: Nested loop example *a)*, and resulting loop tree *b)*. The brackets in the nested loop example represent a high level language iterative construct. The inner loop (smaller bracket) is lexically nested in the outer loop (larger bracket). During the execution of the nested loop example, the outer loop iterates three times. In the first iteration of the outer loop, the inner loop executes for two iterations, creating a definition of X in the second iteration. The definition of X is later used, outside of the scope of the inner loop, in the third iteration of the outer loop. In the loop tree, the solid lines represent pointers to loop instance records, and the dashed lines are pointers to loop iteration records.

of this common loop, from the creating iteration.

We employ the *loop tree* structure to track the program's dynamic loop instances and iterations in a storage efficient manner. Figure 9(b) shows the live well and loop tree resulting from the code example in Figure 9(a). The loop tree contains three node types: loop instance descriptors, loop iteration descriptors, and the base line node.

A *loop instance descriptor* is created for each invocation of a loop in the program. A loop is defined as any subgraph of the CFG in which one basic block, the loop header, dominates all basic blocks up to and including one or more blocks that contain a back edge to the loop header [ASU86]. This definition suffices unless the CFG contains irreducible loops.[4] While irreducible loops can be created in most languages, they are generally not found in practice.[5] In our DDG analyses, we annotate the instruction trace with loop start, end, and continue "signals". The natural loop analysis is performed prior to executing the DDG analyzer. When a "loop start" signal is encountered, a loop instance descriptor is created and linked to the current loop invocation. When a "loop end" signal is encountered, the current loop invocation is reset to the parent of the loop instance that is ending. The loop instance descriptor contains a pointer to the creating loop iteration, a unique brand (used later in a tree search optimization), a tree depth variable (used for tree searches), and a reference count.

All loop tree descriptors contain a reference count indicating how many other descriptors and value records point to it. Since a program can create virtually an unlimited number of loop instances, with even more iterations, it is imperative that loop tree storage be released as soon as it is known to be unreferenced by any value in the live well. We employ a reference counting reclaimation scheme because the extent of the values created in the program, and thus referencing the loop instance records, does not follow any program structure; as a result, a loop descriptor can become free at any time during analysis. Using reference counts, we just reclaim the loop tree storage when its reference count becomes zero.

The *loop iteration descriptors* are linked to the loop instance in which they are contained. Each occurrence of "loop start" or "loop continue" signals cause the creation of a loop iteration descriptors in the loop tree. The loop iteration descriptors contain a pointer to the loop instance that created this iteration, an iteration number, and also a reference count.

Although the loop instance descriptor and loop iteration descriptors could be coalesced, they are kept separate to reduce the size of the loop tree. Because the information in the loop instance descriptor is larger than the pointer used to reference to it, less memory is used.

Whenever a value is referenced, an edge is created in the DDG. The control distance of the edge is computed by walking the loop tree, starting at the iteration descriptor of the use and definition of the value, until a common (loop instance) ancestor is found. The loop distance is then computed by subtracting the iteration number of the definition from the iteration number of the use. If the base line node is reached, the definition and use of the value are not contained within any common loop, and the loop distance is zero. We use two optimizations to speed up the tree walk: depth counts, and branding.

Each loop instance descriptor contains its depth with respect to the base line node. When walking the tree, to find a common ancestor, we can immediately determine which reference to walk up the tree first, *i.e.,* the deeper reference.

The top level loops in the program form a forest of loop trees, since the base line node is not a real loop. Each top level loop tree is assigned a unique numeric "brand" which is then stored in the loop instance descriptors. Since a use and definition must reside in the same top level loop tree, we can quickly determine the loop distance to be zero if the create and use loop instance descriptors have different brands.

The storage required to store the loop tree is bounded to a value proportional to the size of the program's active name space. Even though the number of created loop instance and iteration descriptors is unbounded, our reference counting scheme ensures that there can be no more active loop iteration descriptors than the number of values in the live well. Each value in the live well can point to (and thus increment the reference count of) at most one loop iteration descriptor, and the loop iteration descriptors can (in the worse case) each point to one at most one loop instance descriptor. Since the size of the live

[4]One "loose" definition of an irreducible loop is: a loop which does not contain jumps into the middle from outside of the loop; a reducible loop can only have entries at the loop header. See [ASU86] for a formal definition.

[5]The only SPEC benchmark containing irreducible loops is *spice2g6*. The problem was rectified by rewriting the offending routine in a more structured manner.

well is proportional to the size of the active name space, so too will the loop tree in the worse case. In practice, values in the live well will often point to the same iteration descriptors, thus the actual storage requirements will be much less.

The procedure by which loop distances are computed can be easily adapted to measure "call distance". Recall, that a call distance is the distance, on the call stack, between a definition of a value and its use. The call tree is similar to the loop tree; except only the instance descriptors are required since procedures do not iterate. To compute the call distance, locate the first common ancestor on the call tree. This is the procedure in which the definition of the value occurred (although it may have been in a nested procedure instance). The depth of the value use minus the depth of the common ancestor node is the call distance. The branding optimization does not apply to this computation because the call tree does not form a forest of trees.

The storage required to store the call tree is also bounded to a value proportional to the size of the active name space (for the same reason that the loop tree). And in practice, it is significantly smaller than the loop tree.

Having presented the methodologies used to perform unconstrained DDG extraction, scheduling, and analysis, we can now extend the described methodology to also support more restrictive issue models and limited physical resources. The following sections also discuss the many difficulties in their handling.

3.4 Control Scheduling

Tetra can explore the impact of two processor issue models, one based on the control flow graph (CFG), and one based on the control dependence graph (CDG). The first model, CFG issue, is the traditional issue model. Operations may execute as soon a the most recent (in trace order) branch instruction is resolved. The model also employs a speculation mechanism describing whether or not a particular edge in the CFG is speculated correctly, and thus does not constrain the DDG. A mis-speculated edge inserts a control barrier into the DDG such that no later instructions can be placed earlier in the DDG than the mis-speculated branch instruction. Figure 10(a) shows the DDG of Figure 1(c) after CFG control scheduling. In this example, no speculation was employed, so operations `I3`, `I5`, `I8`, `I10`, and `I14` all insert control barriers into the schedule.

The CFG control scheduling algorithm is shown in Figure 11. The algorithm implements the functions `ControlSchedule()` and `UpdateBarriers()`, which are called from the main scheduler loop after determining the unconstrained placement of the operation (see Figure 7). `ControlSchedule()` simply constrains the operation such that it is later than the level of the last control barrier (stored in the variable `lastBarrier`). `UpdateBarriers()` needs to update `lastBarrier` if the instruction passed to it creates a control barrier. `UpdateBarriers()` is called after the passed instruction has been control and resource scheduled. The procedure first checks if the instruction is a conditional branch (using `CondBranch()`), and if so, attempts to speculate past it in the call to `Speculate()`. `Speculate()` is a processor model dependent routine that implements the speculation mechanism used, *e.g.,* two level adaptive, n-bit counter, static predictor, n-percent correct, etc. If the speculation fails, a control barrier is inserted into the execution graph by updating the constraint variable `lastBarrier`. It is reset to the level of the mis-predicted conditional branch (in `schedLevel`). All future operations will be placed after the mis-predicted conditional branch.

The second control scheduling model, *CDG scheduling*, issues operations based on the more relaxed control model as specified by the control dependence graph. (See [FOW87] or [CFR$^+$91] for a complete description and formal definition.) CDG scheduling allows operations to issued as soon as their surrounding predicate is resolved (or correctly speculated). An instruction is said to be *control dependent* upon its surrounding predicate, because it is this predicate that ultimately determines if the instruction is issued. In Figure 10(b), the DDG of Figure 1(c) has been control scheduled based on the control dependence graph rather than the control flow graph (as is shown in Figure 10(a)). Note how the operations `I6` and `I12` are "elevated" in the scheduled DDG to just after their surrounding predicate, that is, operations `I3` and `I8`, respectively. Contrast this to the CFG scheduling case in Figure 10(a), where `I6` and `I12` are control dependent on `I5` and `I10`, respectively. This mobility is allowed because the operations `I6` and `I12` post-dominate the branches `I5` and `I10`, and thus their issue is invariant of any decisions that they make.

The CDG model actually appears to be multi-threaded with respect to the CFG model because the

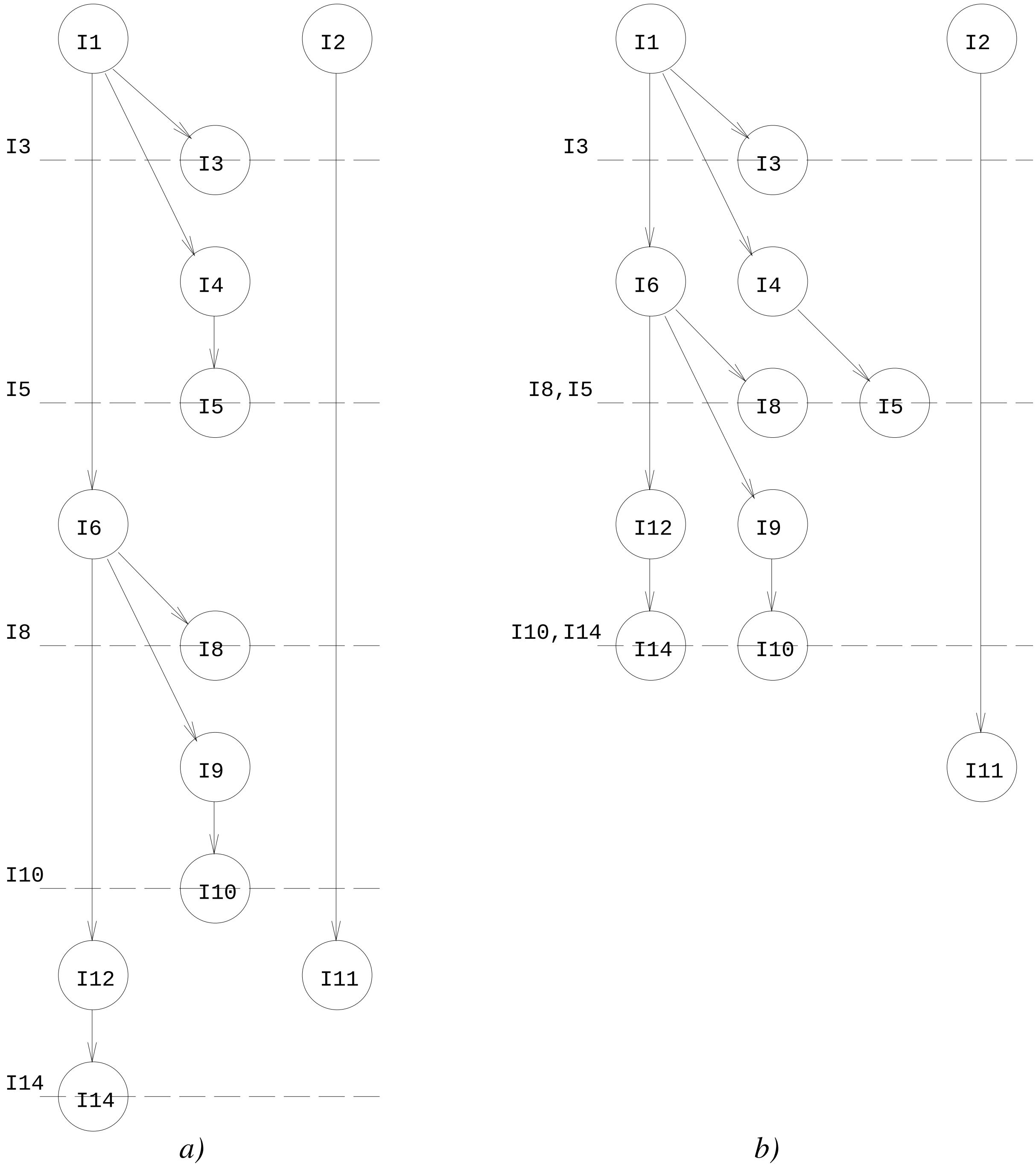


Figure 10: CFG control constrained schedule *a)*, and CDG control constrained schedule *b)*. All schedules are performed on the DDG of Figure 1(c). The solid lines represent flow dependencies, and the node labels correspond to an instruction instance in Figure 1(b). The first control constrained schedule, *a)*, uses a control flow graph (CFG) scheduling model with no speculation. The second control constrained schedule, *b)*, uses a control dependence graph (CDG) scheduling model with no speculation. The control barriers produced by the branch instructions are shown as horizontal dashed lines and are labeled with the instruction (or instructions) creating the control barrier.

```
var
    lastBarrier: Level;

function ControlSchedule(inst: Inst, earliest: Level): Level;
begin
    ControlSchedule := MAX(earliest, lastBarrier);
endfunc

procedure UpdateBarriers(inst: Inst, schedLevel: Level);
begin
    if CondBranch(inst) then
        if NOT Speculate(inst) then
            lastBarrier := schedLevel;
        endif
    endif
endproc
```

Figure 11: CFG Control Scheduling Algorithm.

program can now be simultaneously executing in multiple, non-contiguous areas of the program. In our research, we have found that parallel operations in horizontal slices of the execution graph are indeed non-local, *i.e.,* the independent operations are coming from different areas of the program, thus there is merit in examining this issue model.

The CDG issue model can be exploited either by the hardware or the compiler, or both. For example, Beckmann [BP92] has proposed hardware support for multithreaded execution of serial programs. The execution conditions that fire off threads are based on the CDG rather than the more restrictive CFG. Compiler examples include PDG based scheduling techniques such as region scheduling [AJLS92, GS90]. These techniques implement a CDG based issue model on mundane hardware by applying speculative and non-speculative code motion such that code executes upon resolution of the control dependent predicate, rather than the preceding branch.

Figure 12 shows the CDG based control scheduling algorithm. `ControlSchedule()` constrains the operation to the time of the last control barrier as defined by the CDG. This is either:

i. the level at which last control dependent branch was resolved,

ii. or, if this instruction is not control dependent on any instruction in the current procedure, the level at which the current procedure began execution.

To compute the DDG level for case *i*, the procedure first calls `LastCDNode()` which returns the address of the last executed branch upon which `inst` is control dependent.[6] We employ the computationally efficient dominance frontier technique to annotate all the conditional branches upon which a basic block is control dependent [CFR+91]. This involves first computing the post-dominator tree [LT79]. And then, for each node in the CFG, a depth first traversal is used to identify the nodes in the CFG where some basic block no longer post dominates all of its successors. This is the *dominance frontier* of the CFG node. The control dependent branches are contained in the basic blocks of the dominance frontier. In our DDG analyzer, we annotate the trace stream at the beginning of each basic block with a list of addresses of the basic blocks on which the current one is control dependent (this need only be performed at the beginning of each basic block, as all instructions in a basic block will be control dependent on the same branches); `LastCDNode()` uses this list to find the correct control dependent branch.

The barrier stack (in variable `barrierStack` records the issue time of each procedure in the current call stack. Upon a call to a procedure, the control barrier level of the calling instruction is pushed onto the `lastBarrier` stack. This is later used to schedule instructions that have no statically defined control

[6]The construction of the CDG will often result in a basic block being control dependent upon a number of different conditional branches or none. If dependent on multiple branches, the controlling predicate is always the latest to have executed in the current procedure instance.

```
var
    lastBarrier: MAPPING from Address to Level;
    barrierStack: STACK of Level;

function ControlSchedule(inst: Inst, earliest: Level): Level;
begin
    if LastCDNode(inst) == NONE then
        ControlSchedule := MAX(earliest, barrierStack.TOP);
    else
        ControlSchedule := MAX(earliest, lastBarrier[LastCDNode(inst)]);
    endif
endfunc

procedure UpdateBarriers(inst: Inst, schedLevel: Level);
begin
    if CondBranch(inst) then
        if NOT Speculate(inst) then
            lastBarrier[inst.pc] := schedLevel;
        endif
    endif
    if CallInst(inst) then
        barrierStack.PUSH(lastBarrier[LastCDNode(inst)]);
    endif
    if ReturnInst(inst) then
        barrierStack.POP;
    endif
endproc
```

Figure 12: CDG Control Scheduling Algorithm.

Scheduling Algorithm		Time Complexity		Space Complexity	Shown to always out perform	Schedule non-deterministic latencies?
History		Find	$O(t)$	$O(t)$	List – best fit List – first fit	No, degrades to List – best fit
		Delete	—			
		Insert	$O(l)$			
		Asymp	$O(t)$			
List	Best Fit	Find	$O(\log p)$	$O(p)$	List – first fit	Yes, with check-out/check-in
		Delete	$O(\log p)$			
		Insert	$O(\log p)$			
		Asymp	$O(\log p)$			
	First Fit	Find	$O(1)$	$O(p)$		Yes, with check-out/check-in
		Delete	$O(\log p)$			
		Insert	$O(1)$			
		Asymp	$O(\log p)$			
Round Robin		Find	$O(1)$	$O(p)$		Yes, with check-out/check-in
		Delete	$O(1)$			
		Insert	$O(1)$			
		Asymp	$O(1)$			
Random		Find	$O(1)$	$O(p)$		Yes, with check-out/check-in
		Delete	—			
		Insert	—			
		Asymp	$O(1)$			

Table 1: Comparison of the scheduling heuristics. All time and space complexities are written with respect to either the trace size, t, the number of schedulable resources, p, the maximum deterministic latency, l, or a constant, 1. The "Asymp" time complexity refers to the asymptotic time complexity of managing the resource heap, and is bound to the most costly individual operation. Space complexity refers to just the resource heap nodes, and does not include the live well. Non-deterministic latencies are caused by the scheduling of physical memory resources. Their latencies are not known at the time they are scheduled, therefore a check-out/check-in scheduling mechanism must be employed.

dependencies (case *ii*). When a procedure returns, the issue level of that procedure is popped off of the barrier stack.

3.5 Resource Scheduling

The next phase of scheduling, resource scheduling, serves to further constrain the DDG due to functional unit and storage limitations.

Since generating an optimal schedule for a DAG in the presence of resource limitations is an NP-complete problem, we must employ a scheduling heuristic. *Tetra* implements five scheduling heuristics which exhibit varying degrees of storage and space complexity during execution, and in return, provide equally varying degrees of performance with respect to an optimal schedule. These are: history schedule (HISTORY), list schedule–best fit (LIST-BF), list schedule–first fit (LIST-FF), round robin (RND-RBN), and random (RANDOM). Table 1 summarizes the five resource constrained scheduling heuristics.

Because the heuristic schedules only approximate the optimal schedule, a question naturally arises: how close to optimal is the heuristic schedule? This question is typically answered by generating what is called the *competitive ratio* [GGJ78] for the given heuristic. A competitive ratio is a numeric upper bound showing, for the worse case, how much longer the resulting heuristic schedule is compared to the length of the optimal schedule for the same problem instance. Using I to denote a problem instance, the notation $OPT(I)$ represents the length of the optimal schedule for I, and $HEUR(I)$ is the length of the heuristic schedule. The competitive ratio for the heuristic is:

$$max \left\{ \frac{HEUR(I)}{OPT(I)} \right\} \; \forall I$$

Since we cannot hope to examine all I, one must compute $HEUR(I)$ as a non-naive lower bound and

$OPT(I)$ as an upper bound, thus producing a result that is likely more conservative than the true competitive ratio of the heuristic. Advancing the state-of-the-art for a particular heuristic entails finding better lower bounds on $HEUR(I)$, thus further tightening the accuracy of the competitive ratio. Computation of a non-naive competitive ratio for heuristics which schedule arbitrary DAGs in the presence of resource constraints is still an unsolved problem. We do not attempt to solve it here, rather we have proven (in Appendix A) a relative order on the performance of each of the presented heuristics. This ordering is summarized in Table 1 and in Figure 21 in Appendix A. To summarize those results, the relative ordering is such that as one spends more processor and memory resources on the scheduling heuristic, at least as good or better schedules will be generated.

There are two underlying assumptions that we adopted when selecting heuristics for scheduling limited resources. First, operations are always scheduled in the order that they occur in the trace (*i.e.*, we apply an *on-line* scheduling heuristic). This assumption is required to make the heuristic solution reasonable to implement and execute. Second, we always schedule a particular resource (*e.g.,* registers, physical memory, functional units) from a common pool. For example, when scheduling register resources, the resource pool is not associated with a particular register, but rather with all the registers. While this makes the implementation of the heuristics somewhat more complicated, it yields much better schedules. It is also more representative of compiler and dynamic processor scheduling techniques.

3.5.1 History Schedule

The *history schedule* heuristic is the most powerful (and expensive) because it retains complete history of all resource allocation, and thus it can *back schedule*. Back scheduling allows the scheduler to allocate a resource *earlier* in time than the previous allocation of that resource. While it may seem odd that we are scheduling a resource in the past, it is justifiable. The purpose of our analysis is to evaluate the performance of a program under an abstract processor and compiler model – execution models that aggressively exploit fine-grain serial program parallelism. It is unlikely that, if built, these processors would use the same instruction presentation, *i.e.,* it is likely that the compiler would reorganize the instruction issue order such that it was closer to a breadth first visitation of the unconstrained, scheduled DDG. History scheduling allows *Tetra* to generate the best possible schedule independent of the presentation of the instructions.[7]

Because back scheduling requires deterministic latency for scheduled operations (we are filling a hole in the schedule, so the allocation request must be finite and known), we cannot use this heuristic to build schedules with limited physical storage.

Figure 13 shows the `ResourceSchedule()` function, called from Figure 7, for the history scheduling heuristic.

The variable `resHistory` holds the complete resource history for the resource pool. A different `resHistory` variable will be required for each pool of resources. An entry at index i in the array indicates how many resource instances have been allocated to the level i in the scheduled DDG. Each entry in `resHistory` is bounded to `MAX_RESOURCES`, the total number of resources in the resource pool. Since each level of the DDG is represented in `resHistory`, the array must have at least `CRITICAL_PATH_LENGTH` entries, where `CRITICAL_PATH_LENGTH` is the length of the program's critical path.

The algorithm scans `resHistory` for a slot of width `PLatency(inst)` with sufficient resources. When found, it then increments the array entries for these levels. By clamping the entries to `MAX_RESOURCES`, we can ensure that any resource is allocated in any one level no more than once. `PLatency(inst)` is the pipeline fill latency of the operation, that is, how long the issue of the operation will delay the pipeline. After `PLatency(inst)` DDG levels, another operation can be inserted into the resource pipeline. For non-pipelined resources, like memory, the `PLatency(inst)` is equal to the total resource allocation latency.

In Figure 14(c), a DDG is scheduled using the history schedule heuristic. In this example, the operations `I6` and `I9` were back scheduled.

The history schedule heuristic performs very well, building constrained DDG schedules with very high resource utilization. Yet, the algorithm is very processor and storage greedy. In the worse case, every

[7] While we have not proven this, we believe that an on-line HISTORY scheduler produces schedules comparable to an off-line LIST scheduler. The latter is typically used by compilers to perform instruction scheduling.

```
var
    resHistory: ARRAY 0..CRITICAL_PATH_LENGTH] of integer;

function ResourceSchedule(inst: Inst, earliest: Level): Level
begin
    i, j: integer;
    found: boolean;

    for i := earliest to CRITICAL_PATH_LENGTH do
        found := TRUE;
        for j := i to i + PLatency(inst) - 1 do
            if (resHistory[j] >= MAX_RESOURCES) then
                found := FALSE;
            endif
        endfor
        if (found) then
            for j := i to i + PLatency(inst) - 1 do
                resHistory[j]++;
            endfor
            ResourceSchedule := i;
            return;
        endif
    endfor
endfunc
```

Figure 13: History Scheduling Heuristic.

level of the DDG could have to be searched to locate a slot with sufficient resources, making the time complexity of the algorithm $O(t)$, where t is the length of the trace and the worse case critical path length. The storage complexity of the algorithm is also $O(t)$, since it is conceivable that the last operation may be scheduled in any slot of the resource history array. Thus, we must keep the entire history for the entire analysis.

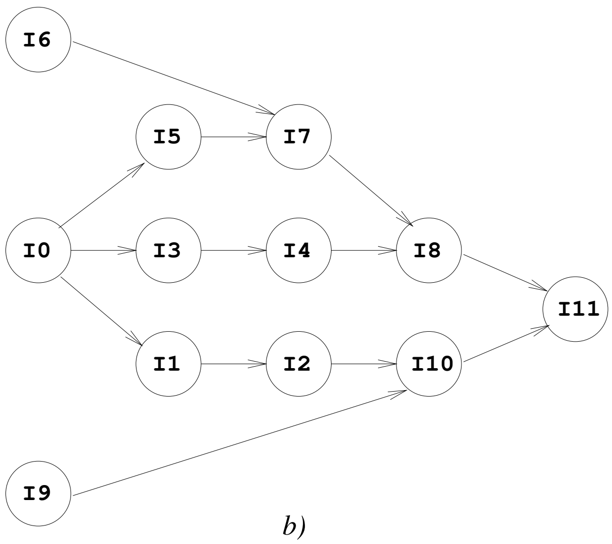
It is possible to release some resource history if the base level (in `baseLevel`) is advancing. This will occur when control barriers and system calls are encountered (with conservative side-effect handling). In *Tetra*, we take advantage of these cases, still in the worse case the time and space complexities are $O(t)$.

When the analyzed program executes many instruction and has little program parallelism (thus a long critical path), the resource history requirements will soon become too great for any reasonable machine, particularly with respect to storage requirements. In this event, we can approximate the resource history, by increasing the granularity of the resource history slots. For a granularity of n, each element of the resource history array will track resources for n contiguous levels of the scheduled DDG, see Figure 15. Using this approximation, it is only possible to limit the average resource demand in the area represented, *i.e.* for a granularity of n with at most p resource instances available, a slot can contain at most $n * p$ operations. The exact form of the DDG in the slot is not known. DDG levels can certainly have more then n operations in one level – which if re-scheduled with a granularity of 1, could require more than n levels to execute with a maximum of p resource instances.

Figure 15 shows how to compute the maximum error when using a resource history array with granularity n. The worse case path length extension (in a single slot) arises when the actual schedule contains one operation on all levels, $n-1$ operations, except one which contains all the remaining operations, $(n*p)-(n-1)$ operations. The resulting error is due to the $(n-1)*(p-1)$ operations that must be rescheduled because they cannot be serviced with p resource instances. Hence, the correct schedule could extend as much as $\frac{(n-1)*(p-1)}{p}$ DDG levels. This is the worse case behavior where all operations are dependent on the operation in the next level, *i.e.,* all operations to be rescheduled are on the critical path. The actual path length is thus extended from n to $n+\frac{(n-1)*(p-1)}{p}$ which converges to $2n$ for large values of n and p. The maximum error is thus always less then 100%, and this for a very atypical worse case – typical executions will have a much smaller path length error.

I0	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10	I11

a)



b)

History:

	0	1	2	3	4	5	6	7
p3	I9	I5	I7					
p2	I6	I3	I4	I10				
p1	I0	I1	I2	I8	I11			

List Schedule – Best Fit:

	0	1	2	3	4	5	6	7
p3		I5	I6					
p2		I3	I4	I9	I10			
p1	I0	I1	I2	I7	I8	I11		

List Schedule – First Fit:

	0	1	2	3	4	5	6	7
p3			I2			I8		
p2		I1	I5		I7		I11	
p1	I0	I3	I4	I6	I9	I10		

Round Robin:

	0	1	2	3	4	5	6	7
p3			I2	I5		I8	I11	
p2		I1	I4		I7	I10		
p1	I0	I3	I6	I9				

c)

Figure 14: Instruction trace *a)*, topologically sorted, unconstrained dynamic dependence graph *b)*, and resulting functional unit constrained schedules for each of the presented scheduling heuristics *c)*. For each heuristic, the instructions are scheduled in the trace order, all operations take unit time, and any of the three functional units are capable of servicing any instruction. Schedule charts show the issue time of a scheduled instruction on the horizontal axis, and the functional unit used (p1, p2, or p3) on the vertical axis.

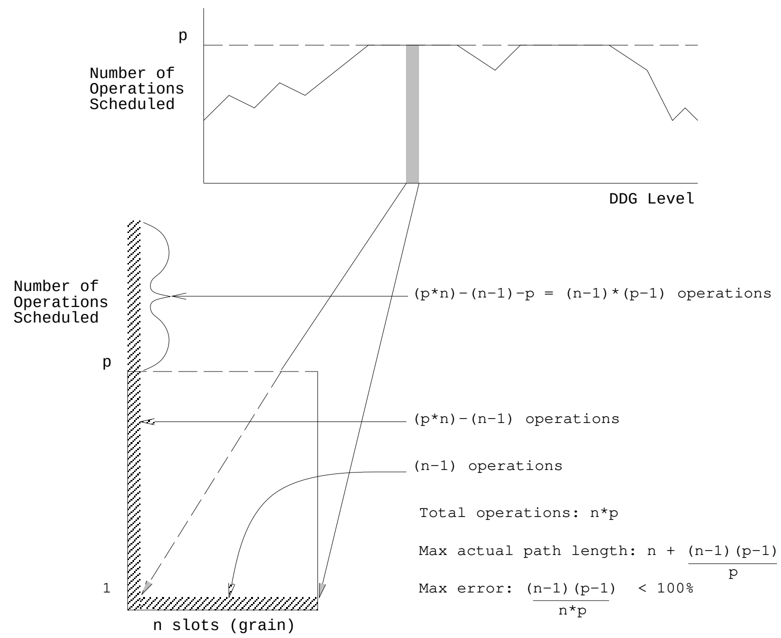


Figure 15: Maximum error computation for history scheduling heuristic. The top graph is a complete parallelism profile for a resource constrained schedule of a program's execution. For this schedule, at most p resources are available in any single DDG level, and the resource history buffer has a granularity of n DDG slots per buffer slot. The lower graph is a magnified reconstruction of the shaded slot from the top parallelism profile. The hashed region of the lower graph represents the *actual* form of the DDG.

3.5.2 List Schedule–First Fit and Best Fit

The list scheduling heuristics reduce time and storage complexities by not allowing back scheduling. Without back scheduling, there is no need to keep complete resource history; instead, all that must be stored is the level in which each resource instance is next available for scheduling. The list scheduling heuristics then select an appropriate resource instance based the state of the resource heap and the earliest time at which the operation to schedule can execute.

The two list scheduling heuristics are differentiated by how they select resource instances. The *list schedule–first fit* heuristic always selects the earliest available resource instance from the resource heap. Figure 16 shows the pseudo-code for the list scheduling–first fit heuristic. Figure 14(c) shows an example DDG scheduled with the list schedule–first fit heuristic. Initially the resource heap contains p resource instances, where p is the total number available for scheduling, and all are available for scheduling at time 0.

The algorithm employs a priority queue to manage the resource heap. The priority queue, `resHeap`, support two operations: DELETE_MIN and INSERT. DELETE_MIN finds the earliest available resource instance and deletes it off the priority queue. Next, the resource is scheduled at either the time the resource instance is available or the earliest it can be scheduled, based on input and control constraints. Finally, the resource instance is re-inserted, using the INSERT operation, into the resource heap $PLatency(inst)$ time units after it is scheduled, as this is when that resource instance will next be available for use by another operation.

In our DDG analyzer, we implement the priority queue with a Fibonacci heap [FT87]. We selected this structure because it is one the fastest know priority queue implementations. Its asymptotic time complexity is $O(1)$ for access to the minimum value entry (its the root of the heap), $O(1)$ for arbitrary value insertions, and $O(\log p)$ for deletions.

We can improve on the first fit resource selection strategy. The earliest available resource instance may be available much earlier than an operation can use it. Other scheduling considerations, such as input availability and control constraints can force the earliest available resource instance to wait idle for long periods of time before it is used to execute the operation. Because list scheduling does not support

```
var
    resHeap: PRIORITY-QUEUE of Level;

function ResourceSchedule(inst: Inst, earliest: Level): Level;
begin
    schedLevel: Level;

    schedLevel := resHeap.DELETE_MIN;
    schedLevel := max(earliest, schedLevel);
    resHeap.INSERT(schedLevel + PLatency(inst));
    ResourceSchedule := schedLevel;
endfunc
```

Figure 16: List Schedule–First Fit Heuristic.

```
var
    resHeap: BINARY-TREE of Level;

function ResourceSchedule(inst: Inst, earliest: Level): Level;
begin
    schedLevel: Level;

    schedLevel := resHeap.DELETE_MAX_PRIOR(earliest);
    schedLevel := max(earliest, schedLevel);
    resHeap.INSERT(schedLevel + PLatency(inst));
    ResourceSchedule := schedLevel;
endfunc
```

Figure 17: List Schedule–Best Fit Heuristic.

back scheduling, this idle period results in lost scheduling opportunities, which in turn reduces overall resource utilization. To reduce this idle resource time, *list schedule–best fit* instead selects the resource instance available closest in time and before the time at which the operation must execute. If all resource instances are available after the time at which the operation must execute (based on previous scheduling constraints), the first available resource instance available is selected. Figure 17 shows the pseudo-code for the list scheduling–best fit heuristic. Figure 14(c) shows an example DDG scheduled with the list schedule–best fit heuristic.

Since selecting the best fitting resource instance is very inefficient with a priority queue ($O(p)$, where p is the number of schedulable resource instances), we instead employ a binary tree. The binary tree nodes are sorted by the time each resource instance is next available for scheduling. The DELETE_MAX_PRIOR function selects the right-most node that is left of `earliest`. If no tree nodes are left of `earliest`, the left-most node is returned.

In *Tetra*, we use an AVL binary tree. This balanced binary tree structure's asymptotic time complexity is $O(\log p)$ for selecting the resource instance, $O(\log p)$ for arbitrary value insertions, and $O(\log p)$ for deletions.

The list schedule (first fit or best fit) storage complexity is always $O(p)$, where p is the number of schedulable resource instances, since there are p resource instance records in the resource heap. The algorithms requires no other storage. The asymptotic time complexity of either algorithm is $O(\log p)$, since the asymptotically dominating operation is deleting the resource instance from the heap, which is $O(\log p)$ for both. While the aggregate time complexity is the same for the list schedule best fit and first fit heuristics, the first fit search and insert components are much cheaper. This results in a significant time savings for smaller values of p (which we find is the typical case).

If the total resource latency is not known, *e.g.,* in the case of memory resource instances, a check-

```
var
    resHeap: QUEUE of Level;

function ResourceSchedule(inst: Inst, earliest: Level): Level;
begin
    schedLevel: Level;

    schedLevel := resHeap.DELETE_TAIL;
    schedLevel := max(earliest, schedLevel);
    resHeap.INSERT_HEAD(schedLevel + PLatency(inst));
    ResourceSchedule := schedLevel;
endfunc
```

Figure 18: Round Robin Scheduling Heuristic.

out/check-in strategy must be employed instead. With check-out/check-in, a scheduled resource instance is left out of the resource heap until it is know to be free, at which point it is checked-in to the resource heap using the resource heap INSERT function. The resource is marked as available immediately after its last use.

There are two caveats that must be observed when generating storage constrained resource schedules. First, resource constrained schedules should be generated from a single unified resource pool. To separate the allocation pools into register and memory resources would certainly give dubious results, as there is no guarantee that the compiler would allocate the same values to registers and memory, especially under vastly different processor models. If multiple pools are desired, then the allocations to the register or memory pools should be based on the frequency of use and lifetime of the residing value. Since this information is only known after release of the storage resource, good storage constrained schedules are difficult to attain. Second, since the extent of the storage use is not known when it is allocated, it cannot be de-allocated until it is re-used. This is the point at which the storage is re-assigned a new value, and in the typical case, the actual last use is much earlier than the time of re-assignment. Consequently, the resource will sit idle for a period of time. Generally, storage constrained schedules without prescient knowledge are very conservative.

3.5.3 Round Robin and Random Schedules

The round robin and random schedules are included because, while they also require $O(p)$ storage, they both schedule operations in $O(1)$ time. Thus their performance must be studied if the use of the better, more computationally expensive scheduling heuristics is to be justified.

The *round robin* scheduler (also commonly referred to as a FIFO scheduler) is detailed in Figure 18. It simply removes the next resource from the tail of a queue, that is, the least previously used resource, and assigns it to the operation. The resource instance is re-inserted into the queue at the head. The resource will not be scheduled again until all other resources have been scheduled. The queue is initially filled with p resource instances.

The *random* scheduler, see Figure 19, is even simpler. It randomly selects the next resource from all existing resources.

4 Tetra: A Tool for DDG Analysis

Tetra is our tool for performing dynamic dependence analysis. It implements all the algorithms described in Section 3 as well as a number of run time and storage optimizations described here. *Tetra* is freely distributable – we encourage practitioners and researchers alike to use it or extend it as they see fit. See the epilogue of this report for distribution details.

Figure 20 shows the dynamic dependence analysis framework that we employ. There are two phases to analysis. In the first phase, trace generation, we use *QPT* (Quick Profiler and Tracer [BL92]) to re-write

```
var
    resHeap: ARRAY [0..MAX_RESOURCES-1] of Level;

function ResourceSchedule(inst: Inst, earliest: Level): Level;
begin
    schedLevel: Level;
    resIndex: integer;

    resIndex := random(MAX_RESOURCES);
    schedLevel := resHeap[resIndex];
    schedLevel := max(earliest, schedLevel);
    resHeap[resIndex] := schedLevel + PLatency(inst);
    ResourceSchedule := schedLevel;
endfunc
```

Figure 19: Random Scheduling Heuristic.

the program executable in such a way that it creates a trace file when run. Normally, we do not need to write the trace file to the disk, we can instead "pipe" this information directly into *Tetra*. If multiple analyses are to be run on the trace file, it is generally faster to save the trace to disk.

In the second phase of analysis, we use a trace regeneration program (also generated by *QPT*) to recreate the original trace along with a number of special "signals" added. The added signals indicate: memory accesses, loop activity, control dependent instruction addresses, function calls, malloc/free calls, source line numbers, and non-local gotos. *Tetra* interfaces to *QPT* via a linked call-back function.

Our analysis framework also supports *CONDOR* execution. *CONDOR* is a collection of software tools and libraries that allow programs to execute remotely on homogeneous architectures, but still access all the program's local resource (via a network layer system call interface) [BLL92]. Using *CONDOR* we are able to save a single trace file, and then "farm" out numerous analyses that will run on idle machines.

Tetra is currently targeted to the MIPs architecture (running on DECstations). It is written in C using Safe-C extensions. Safe-C is a set of C++ templates and macros which performs extensive run-time checking of pointer and array accesses [BA92]. When using Safe-C, a program cannot access out of the bounds or lifetime of any object without notification (*i.e.*, the program dumps core). If Safe-C is disabled, the program will compile with any vanilla 'C' compiler. The impact to the program's run-time is minimal; for a 21 million instruction analysis of *gcc* (compiling `stmt.c` with option -O), run-time bounds checking only increased the run time by 10%.

Tetra is our second DDG analyzer implementation. Our first implementation, *ParaGraph* is described in [AS92]. This newer implementation is nearly two orders of magnitude faster than the previous – even with more capabilities and run time bounds checking code. This overall speedup comes primarily from optimizations to the hash table, the storage allocator, and the trace regenerator.

The primary and most manipulated data structure in *Tetra* is the live well hash table. The hash table is implemented using an array of bucket chains. As the trace creates values, their destination addresses are hashed (using XOR folding) and then inserted into the hash lists. Subsequent references to the value again hashes the address and searches the bucket chain for the correct value record. It is this linear search that dominated the run-time of our first DDG analyzer implementation. Since the hash table is referenced in the same order as addresses are referenced in the executing program, we can expect a large amount of temporal locality in this reference stream. We take advantage of this temporal locality by dynamically reordering the hash table bucket chains, such that the when a data value is referenced, it is moved to the head of the bucket chain. This simple two line change to the original program approximately doubled the speed of the analyzer.

Another simple optimization to the live well was removal of the register value records to a directly addressed array. Because the register space is small, it is faster to store the entire space in a linear array and index the array by the register number. This change resulted in about a 10% performance increase (after adding the dynamic hash re-ordering).

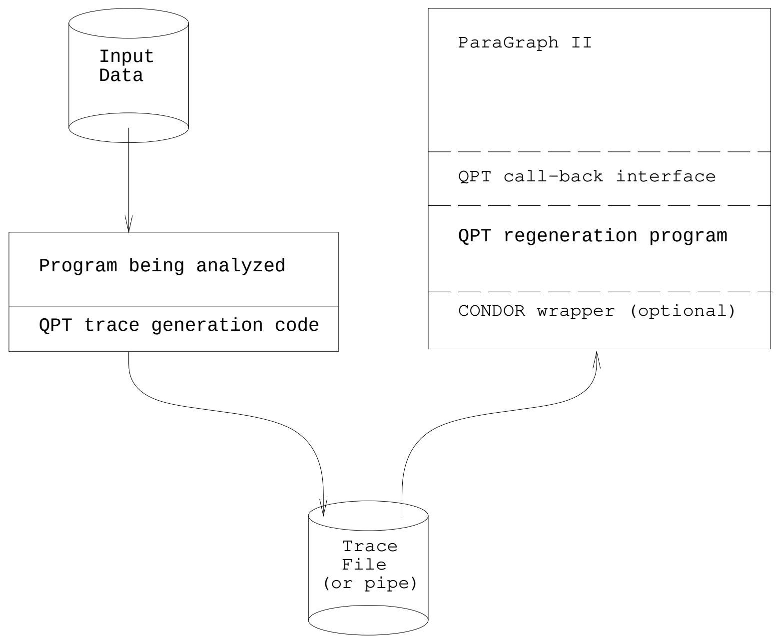


Figure 20: The trace generation, regeneration, and analysis framework. The drum shaped objects represent data files, and the rectangular objects represent executable programs. The directed edges between programs and data files are file I/O paths. A dashed line within a program indicates a procedure call, linked interface. A solid line within a program, *e.g.*, between the program being analyzed and the QPT trace generation code, indicates an object level code transformation.

Another factor that slowed our original implementation significantly was memory thrashing. For analysis of programs with large data sets, the working set of the program would quickly become larger than physical memory. We found that nearly half of allocated memory was unusable – the result of internal fragmentation in the standard Unix malloc package. The typical Unix malloc package is optimized for speed rather than space. These packages typically employ a power of two allocation strategy. If a program requests a block of memory, say 20 bytes in size, the allocate will add a header to this request, now making the needed storage, say 28 bytes. Then the allocator picks a 32 byte block for the node from a list of 2^n size nodes. If the needed node size is not available, a larger block is bisected to fulfill the request. Using this scheme, allocation is indeed very fast, but the storage overhead can be as high as 100%. This overhead need not be born if the needed nodes are all of a fixed size. Such is the case for most allocation in *Tetra*. The primary structures allocated in *Tetra* are value records, loop descriptors, and resource instance records – all of which are individually fixed size. In each case, we employ a generic fixed size heap manager to reduce the storage overhead.

The fixed size heap manager allocates fixed size nodes in large single allocations (chunks) that are very near a power of two in size, thus reducing internal fragmentation from the standard Unix allocator. The heap manager then builds a free list of fixed size nodes from the large allocation. This allocation strategy has a number of benefits. First, because the nodes are allocated *en masse*, they have nearly no per node overhead. A few bytes of overhead is amortized across all the allocated chunks. This has the effect of reducing overall memory requirements. Typically, we see analyses using 40 to 60% less memory with this allocation strategy. The second benefit is increased program spatial locality. While we did not measure this exactly, the analyzer's cache performance is likely better because frequently used data resides in a smaller space – any overheads incurred are moved away from the individual allocations. Another advantage of this strategy is that individual heaps can be deleted in their entirety – there is no need to delete the individual nodes allocated. This is in contrast to a common pool allocator, like malloc, in which an entire class of structures can only be deallocated by freeing each individual allocation (malloc, in fact, does not even support resetting of the heap in any standard way).

Our original analyzer implementation used *Pixie* to generate and regenerate traces; *Tetra* now uses *QPT*. *Pixie* uses no trace compression or minimal witness techniques, thus the resulting traces quickly become larger than that which we can store on our disks. This requires that a pipe be used between the trace generator to the trace regenerator. Our initial implementation's performance was further perturbed because the trace regenerator also piped its output to the DDG analyzer – thus two pipe stages were used to carry information from the program being analyzed to the analyzer. Unix pipes are extremely costly to use, since they rely heavily on system calls. This results a large amount of user to system space copying (and vica versa) as well as costs associated with the many system calls executed. Using *QPT*'s produces much smaller trace files, thus the trace file can be stored to disk (we can easily store 1-2 billion instruction traces on a 250 megabyte disk). This eliminates the trace generator to trace regenerator pipe. The trace regenerator to *Tetra* pipe is eliminated by linking the regeneration program directly to *Tetra*. Not only does the run-time of the program improve, it also spends much less time in the kernel, thus the overall response of the machine is improved.

There were also optimizations we implemented to speed up control scheduling and storage demand profile generation.

Basic block control scheduling optimizes the basic algorithms of Section 3.4 by noting that all instructions in a basic block are always control dependent upon the same branch, independent of the control scheduling methodology employed. As a result, we can determine the last control barrier for all the instructions in the basic block at the beginning of the basic block. Likewise, we need only determine if any control barriers were created at the end of the basic block's execution, as only its last instruction (or second to last in the case of delay slots) could create a control barrier.[8]

Delta distributions provided significant speed up during the generation of storage demand profiles. At first, we incremented the range representing the entire lifetime in the distribution at the time the value became dead. This operation is $O(t)$, where t is the size of the trace. For longer traces, we found that a very large portion of the total execution time was spent updating storage demand profiles. A delta distribution does the same operation in $O(1)$ time by incrementing the slot at which the variable becomes alive, and decrementing the slot at which the variable becomes dead. Each slot in the profile then contains the first derivative (slope) of the storage demand profile. To recreate the storage demand profile, integrate across the entire range – this simply requires summing the delta values and writing the current sum at each slot into the distribution output file.

We close this section by touching a bit on testing. *Tetra* is a very large and complex program (about 25,000 lines of C code). And in an environment where analyses can be billions of instructions in length, it becomes nearly impossible to apply intuition to verify that the output is indeed correct. Hence, a testing strategy needs to be applied. We tested *Tetra* via a parallel trace printing option. If set, this option outputs the execution graph (or an interval therein) to a file after execution completes. That is, first, the instructions in the first level of the execution graph are output, then the instructions in the second level, and so on. While this is not generally applicable to very large traces, it served well for testing. Small programs were constructed such that they exhibited features that would exercise certain codes (*e.g.,* looping code to test the loop distance implementation), and the parallel traces were meticulously checked to ensure 1) the schedule was correct for the specified hardware configuration, and 2) the analysis was correct for the resulting execution graph.

5 Summary

Tetra is a tool for evaluating serial program performance under the resource and control constraints of fine-grain parallel processors. *Tetra*'s primary advantage to the architecture designer is its ability to quickly generate performance metrics for yet to be designed architectures. All the user needs to specify is the capabilities of the architecture (e.g., number of functional units, issue model, etc.), rather than its implementation.

Tetra employs a four step process. First, it extracts a canonical form of the program from a serial instruction trace generated by QPT. It then applies control and resource constraint scheduling to produce an execution graph. The global control and resource constraint scheduling is directed by a processor model specification supplied to the program. Finally, the resulting execution graph is analyzed to evaluate the serial program's performance under the specified architectural model.

[8]This optimization also simplified handling of the delay slot instructions, since the control barriers were computed at the start of the next basic block rather than at the point a conditional branch was detected – this allows *Tetra* to easily enforce the semantics that the instruction after the conditional branch be under the same control constraints as the branch.

The control scheduler supports scheduling based on both the control flow graph and the less restrictive control dependence graph. Both control scheduling models also support speculation using a number of static and dynamic prediction methods. The resource scheduler supports five scheduling heuristics (of varying cost, performance, and capability) which limit the amount of processor (functional unit) and memory resources (physical renaming store) available when building the execution graph.

Once scheduled, *Tetra* provides a number of ways to analyze the program's performance. These include parallelism profiles, storage demand profiles, data sharing distributions, data lifetime analysis, and control distance (branch, loop, and call stack) distributions.

We feel that *Tetra* fills an important niche between intuitive speculation and construction of a complete architectural simulator and compiler. We invite practitioners and researchers to get *Tetra* and apply it to their work. Distribution details are given in the epilogue of this report.

Acknowledgements

We gratefully acknowledge the contributions of the following people. Alain Kägi, for graciously offering to write the Fibonacci heap code, and for his invaluable input. Jim Larus, for writing and supporting QPT. Tom Ball, for his input, and also for his support of QPT. And, Dionisios Pnevmatikatos, Scott Breach, Alvy Lebeck, Babak Falsafi, Steve Reinhardt, and Vikram Adve for their input to this work. This work was supported by National Science Foundation grant CCR-8919635. The author Todd Austin wishes to also express gratitude to his wife and son for their unending support while he was developing *Tetra* and working on this report.

Tetra/QPT Availability

Readers may get the latest *Tetra* source release via anonymous ftp or electronic mail. The anonymous ftp distribution is on "ftp.cs.wisc.edu" in the file "pub/Tetra/tetra-x.tar.Z" ('x' is the version number). For an electronic mail distribution (Unix shar format), send electronic mail to "austin@cs.wisc.edu". The *Tetra* distribution is provided free of charge and with very flexible copyright restrictions. *QPT* is available from Jim Larus (larus@cs.wisc.edu). The licensing agreement and ordering instructions for *QPT* are available from "primost.cs.wisc.edu" in the file "pub/qpt-license.ps.Z". *QPT* is also available in the WARTS distribution (Wisconsin Architectural Research Tool Set); details concerning this distribution are available from "primost.wisc.edu" in the file " ftp/pub/warts-license.ps.Z", or the same information can be obtained by sending mail to "warts@cs.wisc.edu".

References

[ACM88] Arvind, D. E. Culler, and G. K. Maa. Assessing the benefits of fine-grained parallelism in dataflow programs. In *Conference Proceedings of Supercomputing 88*, pages 60–69, November 1988.

[AJLS92] V. H. Allan, J. Janardhan, R. M. Lee, and M. Srinivas. Enhanced region scheduling on a program dependence graph. In *Conference Record of the 25th Annual International Symposium on Microarchitecture*, pages 72–80, Portland, OR, December 1992. Association for Computing Machinery.

[AN89] Arvind and Rishiyur S. Nikhil. A dataflow approach to general-purpose parallel computing. Computation Structures Group Memo 302, MIT, July 1989.

[AS92] Todd M. Austin and Gurindar S. Sohi. Dynamic dependency analysis of ordinary programs. In *Conference Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 342–351. Association for Computing Machinery, May 1992.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[BA92] Scott Breach and Todd Austin. Safe-C: Seat belts for C. available from Todd Austin, austin@cs.wisc.edu, Fall 1992.

[BL92] T. Ball and J. R. Larus. Optimally profiling and tracing programs. In *Conference Record of the 19th ACM Symposium on Principles of Programming Languages*, pages 59–70, Albuquerque, NM, January 1992. Association for Computing Machinery.

[BLL92] Allan Bricker, Michael Litzkow, and Miron Livny. Condor technical report. Technical Report 1069, Computer Sciences Department, University of Wisconsin–Madison, January 1992.

[BP92] Carl J. Beckmann and Constantine D. Polychronopoulos. Microarchitecture support for dynamic scheduling of acyclic task graphs. In *Conference Record of the 25th Annual International Symposium on Microarchitecture*, pages 140–148, Portland, OR, December 1992. Association for Computing Machinery.

[BYP+91] M. Butler, T. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. Single instruction stream parallelism is greater than two. In *Conference Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 276–286. Association for Computing Machinery, May 1991.

[CA88] David E. Culler and Arvind. Resource requirements of dataflow programs. In *Conference Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 141–150. Association for Computing Machinery, May 1988.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and and the control dependence graph. *ACM Transactions on Programming Langauages and Systems*, 13(3):451–490, October 1991.

[CSY90] D. K. Chen, H. M. Su, and P. C. Yew. The impact of synchronization and granularity on parallel systems. In *Conference Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 239–248. Association for Computing Machinery, May 1990.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its uses in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[FS92] Manoj Franklin and Gurindar S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. *Computer Architecture News*, pages 58–67, May 1992.

[FT87] Micheal L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the Association for Computing Machinery*, 34(3):596–615, July 1987.

[GGJ78] M. R. Garey, R. L. Graham, and D. S. Johnson. Performance guarantees for scheduling algorithms. *Operations Research*, 26(1):3–21, January 1978.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[GS90] Rajiv Gupta and Mary Lou Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–431, April 1990.

[Kea74] D. J. Kuck and et al. Measurements of parallelism in ordinary FORTRAN programs. *Computer*, 27:37–46, January 1974.

[Kum88] Manoj Kumar. Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE Transactions on Computers*, 37(9):1088–1098, September 1988.

[Lar91] James R. Larus. Estimating the potential parallelism in programs. In Alexandru Nicolau, David Gelernter, Thomas Gross, and David Padua, editors, *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, chapter 17, pages 331–349. MIT Press, 1991.

[LK78] J. K. Lenstra and A. H. G. Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26(1):22–35, January 1978.

[LT79] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[LW92] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. In *Conference Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46–57. Association for Computing Machinery, May 1992.

[MWC+91] S. K. Mahlke, N. J. Warter, W. Y. Chen, P. P. Chang, and W. W. Hwu. The effect of compiler optimizations on available parallelism in scalar programs. *Preceedings of the 20th Annual International Conference on Parallel Processing*, pages 142–146, 1991.

[NF84] A. Nicolau and J. Fisher. Measuring the parallelism available for very long instruction word architectures. *IEEE Transactions on Computers*, 33(11):968–976, November 1984.

[Nik90] Rishiyur S. Nikhil. The parallel programming language id and its compilation for parallel machines. Computation Structures Group Memo 313, MIT, July 1990.

[SJH89] M. D. Smith, M. Johnson, and M. A. Horowitz. Limits on multiple instruction issue. In *Conference Proceedings of the Third International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 290–302. Association for Computing Machinery, May 1989.

[Tar83] Robert E. Tarjan. *Data Structures and Network Algorithms*, chapter 1.5. Society for Industrial and Applied Mathematics, 1983.

[TF70] G. S. Tjaden and M. J. Flynn. Detection and parallel execution of parallel instructions. *IEEE Transactions on Computers*, 19(10):889–895, October 1970.

[TGH92] K. B. Theobald, G. R. Gao, and L. J. Hendren. On the limits of program parallelism and its smoothability. In *Conference Record of the 25th Annual International Symposium on Microarchitecture*, pages 10–19, Portland, OR, December 1992. Association for Computing Machinery.

[Wal91] D. W. Wall. Limits of instructional-level parallelism. In *Conference Proceedings of the Fourth International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 176–188. Association for Computing Machinery, April 1991.

Appendix A: Relative Performance of the Scheduling Heuristics

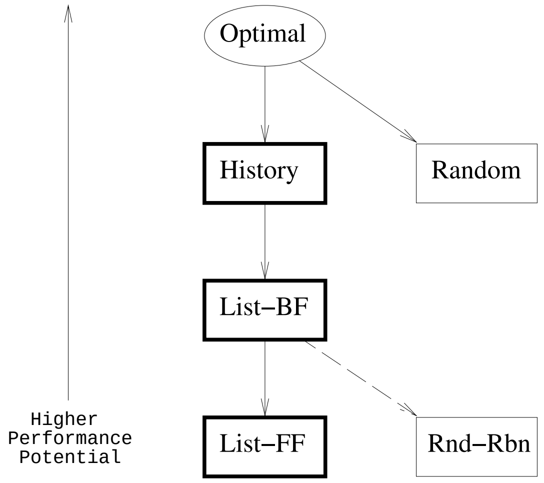


Figure 21: Relative performance of the five scheduling heuristics. The arrows are drawn from the superior schedulers to the inferior. The solid arrows show a performance order proven in the text. The dashed arrow shows a likely, but unproven order. The heuristics drawn with bold borders use a sorted selection method, the others do not.

Figure 21 shows the relative performance of the five heuristics studied in this paper. The arrows are drawn from superior heuristics to inferior heuristics. A heuristic is inferior to another if, for any input, it produces a schedule at least as long (or longer) than its superior counterpart. This relation is transitive, so any descendant of a heuristic is also inferior. The root of the tree is not a heuristic, but rather the optimal schedule, from which all heuristics must descend.

The scheduling heuristics can be classified into two broad categories – those which employ sorted selection methods (shown with bold boxes in Figure 21), and those that do not (shown without bold boxes in Figure 21). Sorted selection methods sort the resource pools by the time when resource instances are next available for scheduling. This sorting allows the schedulers to make reasonably good decisions as to which resource should be scheduled next. The more the heuristic invests into ordering its resource heap, the better decisions it can make. In fact, we can prove this by showing a total order of performance between all the presented heuristics that sort their resource pools.

Claim 1 *Let $HEUR(I,p)$ be the length of the schedule by heuristic $HEUR$ for problem instance I using p resource instances. Problem instance I is a set of operations each requiring one instance from p to execute. All schedules place the operations from I in the same order. The resulting relative performance of the sorted pool scheduling heuristics will be*

$$\textit{HISTORY(I,p)} \leq \textit{LIST-BF(I,p)} \leq \textit{LIST-FF(I,p)} \quad \forall\ I \textit{ and } p.$$

The mnemonics HISTORY, LIST-BF, and LIST-FF, represent schedules by the history, list schedule–best fit, and list schedule–first fit schedulers, respectively.

Proof: Because $\leq$ is transitive, we need only show that for all I and p

$$\text{HISTORY(I,p)} \leq \text{LIST-BF(I,p)},$$

and

$$\text{LIST-BF(I,p)} \leq \text{LIST-FF(I,p)}.$$

For each scheduling heuristic, we define a set, R_i^{HEUR}. This set contains the time at which each schedulable resource instance is next available for scheduling after scheduling the i^{th} operation with heuristic $HEUR$. Thus $|R| = p$, and $0 \leq i \leq T$, where T is the length of the input trace I.

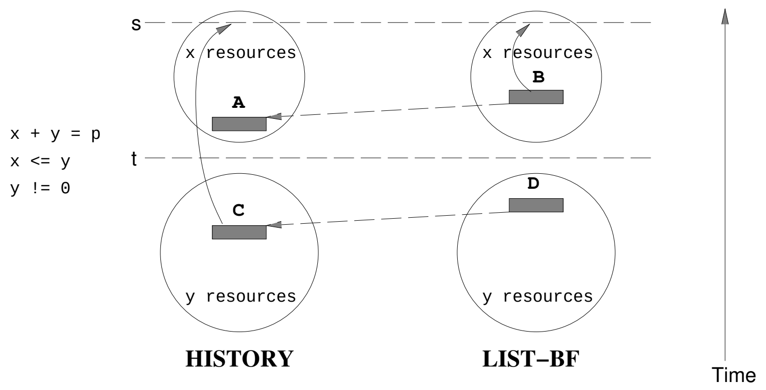


Figure 22: Violating Condition for $|R_i^{HISTORY} < t| \geq |R_i^{LIST-BF} < t|$.

The length of the resulting schedule for heuristic $HEUR$ is $max(R_T^{HEUR})$, where $max()$ is a set function returning the maximum member. In other words, the schedule length is equal to the time is takes for the last resource instance to finish after the last operation is scheduled.

Let $|R_i^{HEUR} < t|$ be the number of resource instances in the set R_i^{HEUR} that are ready to schedule before time t. We prove an order on the performance by showing that for two heuristics, say $BETTER$ and $WORSE$:

$$|R_i^{BETTER} < t| \geq |R_i^{WORSE} < t| \ \forall \ i \ and \ t. \tag{1}$$

To put it another way, as the schedule progresses, the better heuristic will always have as least as many resource instances "lagging behind" those of the inferior heuristic. It follows then that

$$max(R_i^{BETTER}) \leq max(R_i^{WORSE}) \ \forall \ i. \tag{2}$$

For if $max(R_i^{BETTER})$ were greater than $max(R_i^{WORSE})$, there would be p resource instances from the $WORSE$ resource pool before the maximum resource instance of the $BETTER$ schedule – this contradicts assertion (1). Given that assertion (1) holds for all I and p, we can conclude that

$$\text{BETTER(I,p)} \leq WORSE(I, p)$$

since $max(R_T^{BETTER}) \leq max(R_T^{WORSE})$.

Using assertion (1) it is also possible to derive an invariant on the lower bound of values from R_i^{BETTER} and R_i^{WORSE}. That is

$$min(R_i^{BETTER}) \leq min(R_i^{WORSE}) \ \forall \ i. \tag{3}$$

Since there are no resource instances from $BETTER$ before $min(R_i^{BETTER})$, assertion (1) implies the same for $WORSE$. While this invariant is not needed for this proof, we do employ it as a run-time check in *ParaGraph II*.

All that remains is to prove assertion (1) for each of the heuristic pairs mentioned earlier.

$|R_i^{HISTORY} < t| \geq |R_i^{LIST-BF} < t| \ \forall \ i \ and \ t$: We prove this by induction. The basis being

$$|R_0^{HISTORY} < t| = |R_0^{LIST-BF} < t| \ \forall \ t.$$

This holds true as all resource instances are initially available at time 0. Let s be the earliest time that some operation to be scheduled can be executed (s might not be 0 due to previous scheduling constraints). Assuming that

$$|R_i^{HISTORY} < t| \geq |R_i^{LIST-BF} < t| \tag{4}$$

the assertion

$$|R_{i+1}^{HISTORY} < t| \geq |R_{i+1}^{LIST-BF} < t| \tag{5}$$

is only violated in the event that *all* the following conditions hold for some t:

i. $|R_i^{HISTORY} < t| = |R_i^{LIST-BF} < t| \neq 0$, this is the boundary condition at which assertion (5) can fail.

ii. $t \leq s$, this defines a bound on t below which the assertion (5) can fail.

iii. HISTORY schedules a resource instance that was available before time t, and LIST-BF schedules a resource instance that was available after time t, but before or at time s. This condition violates assertion (5) since now $|R_{i+1}^{HISTORY} < t| < |R_{i+1}^{LIST-BF} < t|$ for some $t < s$.

The essence of the conditions is that HISTORY resource instances may only "pass" LIST-BF resource instances when the number of HISTORY resource instances remaining before s is greater or equal to the number of LIST-BF resource instances. Figure 22 shows this condition and why it cannot occur. The shaded boxes in the figure are the resource instances immediately before (C and D) and after (A and B) t. Assertion (4) and condition (i) ensure that if resource instance B is scheduled, there must exist an instance in the HISTORY resource pool that is available after t and at or before B (if the instance did not exist, the first HISTORY resource instance after B would have one more LIST-BF resource instances than HISTORY resource instances before it.) Note that if t is moved, say to t' between C and D, condition (i) no longer holds, as there is one more HISTORY resource instances than LIST-BF resource instances before t' – and performing the action in condition (iii) does not violate assertion (5).

One other subtlety must be examined before we can declare assertion (4) proved. A recurrence exists between the value of s and the placement heuristic. The implication to this proof is that when comparing two heuristics in a step-by-step fashion, we cannot assume that s is the same for both. If one heuristic places an operation later than the other, it may force a later operation scheduled to have a later value of s (due to input values being available at a later time). Fortunately, we can make two observations which will show why assertion (4) can never be violated. First, s is non-decreasing, thus all heuristic will minimally schedule an operation at time s. Second, if $|R_i^{HISTORY} < t|$ becomes larger than $|R_i^{LIST-BF} < t|$, the LIST-BF scheduler was forced to place an operation at a later time than the HISTORY scheduler. This then could force some later s to the LIST-BF scheduler to be larger than the corresponding HISTORY scheduler s. Only the possibility of smaller s values to the LIST-BF scheduler could violate our previous arguments, hence the violating condition cannot occur, and by induction, assertion (4) is proved.

It is the back scheduling capability of the HISTORY scheduler that allows $|R_i^{HISTORY} < t|$ to become greater than $|R_i^{LIST-BF} < t|$. When an operation is back scheduled, $|R_i^{HISTORY} < t|$ is unchanged, while $|R_i^{LIST-BF} < t|$ may decrease in size. Once $|R_i^{HISTORY} < t|$ increases in size, an impassable barrier, as shown in Figure 22, is created.

$|R_i^{LIST-BF} < t| \geq |R_i^{LIST-FF} < t|\ \forall\ i$ *and* t: We prove this by induction. The basis being

$$|R_0^{LIST-BF} < t| = |R_0^{LIST-FF} < t|\ \forall\ t.$$

This holds true as all resource instances are initially available at time 0. Let s be the earliest time that some operation to be scheduled can be executed (s might not be 0 due to previous scheduling constraints). Assuming that

$$|R_i^{LIST-BF} < t| \geq |R_i^{LIST-FF} < t| \tag{6}$$

the assertion

$$|R_{i+1}^{LIST-BF} < t| \geq |R_{i+1}^{LIST-FF} < t| \tag{7}$$

will always be satisfied. We need only examine the case where $t \leq s$, elsewhere, $|R_i^{LIST-BF} < t|$ and $|R_i^{LIST-FF} < t|$ are unaffected. If there are resource instances below t, the LIST-FF scheduler *must* select a resource below t (the earliest available), while the LIST-BF scheduler *may* select a resource below t depending on whether a better fit exists later than t (see Figure 23). Hence, $|R_i^{LIST-BF} < t|$ *may* decrease and $|R_i^{LIST-FF} < t|$ *must* decrease, always leaving $|R_{i+1}^{LIST-BF} < t| \geq |R_{i+1}^{LIST-FF} < t|$.

Once again, we must consider the possibility of the s values of the LIST-FF scheduler becoming larger than those of the LIST-BF scheduler. This event will only serve to increase the possibility

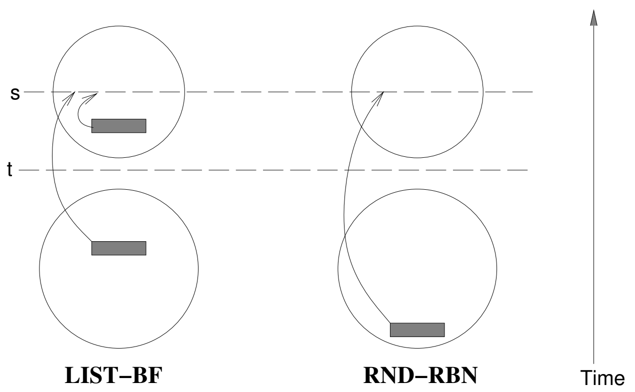


Figure 23: Possible resource placement during a scheduling step of LIST-BF and LIST-FF.

that $|R_{i+1}^{LIST-BF} < t|$ grows larger than $|R_{i+1}^{LIST-FF} < t|$ – hence, by induction, assertion (6) is proved.

Since all the relations have no presuppositions concerning I and p, we can conclude that

$$\text{HISTORY(I,p)} \leq \text{LIST-BF(I,p)} \leq \text{LIST-FF(I,p)} \quad \forall\, I \textit{ and } p. \quad \#$$

Next we examine the unsorted selection heuristics. RANDOM is the easiest to handle because its random selection strategy permits any possible schedule to be constructed.

Claim 2 *No performance order can be shown between RANDOM and any other presented heuristic. All that can be said is that is performs the same or worse than the optimal scheduler.*

Proof: Problem instances can be constructed in which RANDOM performs worse than all the other presented heuristics – such is the case if the RANDOM scheduler selects the *same* resource instance on each step of the schedule. On the other hand, problem instances can also be constructed such that all the other presented heuristics perform suboptimally, but given the correct random sequence, RANDOM *could* produces an optimal schedule. Therefore, no order can be shown between RANDOM and the other heuristics, and the statement is proved. #

Lastly, we examine the performance potential of RND-RBN with respect to the other scheduling heuristics. We do believe after examining the problem closely that RND-RBN will always perform the same or worse than LIST-BF. Numerous attempt to generate a proof of this statement or a counter example have failed. In practice, RND-RBN generally performs much worse than LIST-BF.

Appendix B: Tetra Installation and User's Guide

The following pages are the installation and user's guide for *Tetra* at the time of writing. See the man page file "tetra.1" for the latest version of this document.