Alchemist: A Transparent Dependence Distance Profiling Infrastructure

Xiangyu Zhang, Armand Navabi and Suresh Jagannathan
Department of Computer Science
Purdue University, West lafayette, Indiana, 47907
{*xyzhang,anavabi,suresh*}*@cs.purdue.edu*

***Abstract*—Effectively migrating sequential applications to take advantage of parallelism available on multicore platforms is a well-recognized challenge. This paper addresses important aspects of this issue by proposing a novel profiling technique to automatically detect available concurrency in C programs. The profiler, called Alchemist, operates completely transparently to applications, and identifies constructs at various levels of granularity (e.g., loops, procedures, and conditional statements) as candidates for asynchronous execution. Various dependences including read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW), are detected between a construct and its continuation, the execution following the completion of the construct. The time-ordered *distance* between program points forming a dependence gives a measure of the effectiveness of parallelizing that construct, as well as identifying the transformations necessary to facilitate such parallelization. Using the notion of post-dominance, our profiling algorithm builds an execution index tree at run-time. This tree is used to differentiate among multiple instances of the same static construct, and leads to improved accuracy in the computed profile, useful to better identify constructs that are amenable to parallelization. Performance results indicate that the profiles generated by Alchemist pinpoint strong candidates for parallelization, and can help significantly ease the burden of application migration to multicore environments.**

***Keywords*-profiling; program dependence; parallelization; execution indexing**

I. Introduction

The emergence of multicore architectures has now made it possible to express large-scale parallelism on the desktop. Migrating existing sequential applications to this platform remains a significant challenge, however. *Determining code regions that are amenable for parallelization*, and *injecting appropriate concurrency control into programs to ensure safety* are the two major issues that must be addressed by any program transformation mechanism. Assuming that code regions amenable for parallelization have been identified, techniques built upon transactional or speculation machinery [24], [25], [7] can be used to guarantee that concurrent operations performed by these regions which potentially manipulate shared data in ways that are inconsistent with the program's sequential semantics can be safely revoked. By requiring the runtime to detect and remedy dependency violations, these approaches free the programmer from explicitly weaving a complex concurrency control protocol within the application.

Identifying code regions where concurrent execution can be profitably exploited remains an issue. In general, the burden of determining the parts of a computation best suited for parallelization still falls onto the programmer. Poor choices can lead to poor performance. Consider a call to procedure p that is chosen for concurrent execution with its calling context. If operations in p share significant dependencies with operations in the call's continuation (the computation following the call), performance gains that may accrue from executing the call concurrently would be limited by the need to guarantee that these dependencies are preserved. The challenge becomes even more severe if we wish to extract data parallelism to allow different instances of a code region operating on disjoint memory blocks to execute concurrently. Compared to function parallelism, which allows multiple code regions to perform different operations on the same memory block, data parallelism is often not as readily identifiable because different memory blocks at runtime usually are mapped to the same abstract locations at compile time. Static disambiguation has been used with success to some extent through data dependence analysis in the context of loops, resulting in automatic techniques that parallelize loop iterations. However, extracting data parallelism for general programs with complex control flow and dataflow mechanisms, remains an open challenge.

To mitigate this problem, many existing parallelization tools are equipped with profiling components. POSH [17] profiles the benefits of running a loop or a procedure as speculative threads by emulating the effects of concurrent execution, squashing and prefetching when dependencies are violated. MIN-CUT [14] identifies code regions that can run speculatively on CMPs by profiling the number of dependence edges that cross a program point. TEST [5] profiles the minimum dependence distance between speculative threads. In [23], dependence frequencies are profiled for critical regions and used as an estimation of available concurrency. Dependences are profiled at the level of execution phases to guide behavior-oriented parallelization in [7].

In this paper, we present Alchemist, a novel profiling system for parallelization, that is distinguished from these efforts in four important respects:

1) *Generality.* We assume *no* specific underlying runtime execution model such as transactional memory to deal with dependency violations. Instead, Alchemist pro-

vides direct guidance for safe manual transformations to break the dependencies it identifies.

2) *Transparency.* Alchemist considers all aggregate program constructs (e.g., procedures, conditionals, loops, etc.) as candidates for parallelization with no need for programmer involvement to identify plausible choices. The capability of profiling all constructs is important because useful parallelism can be sometimes extracted even from code regions that are not frequently executed.
3) *Precision.* Most dependence profilers attribute dependence information to syntactic artifacts such as statements without distinguishing the context in which a dependence is exercised. For example, although such techniques may be able to tell that there is a dependence between statements x and y inside a loop in a function *foo()*, and the frequency of the dependence, they usually are unable to determine if these dependences occur within the same loop iteration, cross the loop boundary but not different invocations of *foo()*, or cross both the loop boundary and the method boundary. Observe that in the first case, the loop body is amenable to parallelization. In the second case, the method is amenable to paralelization. By being able to distinguish among these different forms of dependence, Alchemist is able to provide a more accurate characterization of parallelism opportunities within the program than would otherwise be possible.
4) *Usability.* Alchemist produces a ranked list of constructs and an estimated measure on the work necessary to parallelize them by gathering and analyzing profile runs. The basic idea is to profile dependence distances for a construct, which are the time spans of dependence edges between the construct and its continuation. A construct with all its dependence distances larger than its duration is amenable for parallelization. The output produced by Alchemist provides clear guidance to programmers on both the potential benefits of parallelizing a given construct, and the associated overheads of transformations that enable such parallelization.

The paper makes the following contributions:

- Alchemist is a novel transparent profiling infrastructure that given a C or C++ program and its input, produces a list of program points denoting constructs that are likely candidates for parallelization. Alchemist treats *any* program construct as a potential parallelization candidate. The implication of such generality is that a detected dependence may affect the profiles of multiple constructs, some of whom may have already completed execution. A more significant challenge is to distinguish among the various dynamic nesting structures in which a dependence occurs to provide more accurate and richer profiles. We devise a sophisticated online algorithm that relies on building an index tree on program execution to maintain profile history. More specifically, we utilize a post-dominance analysis to construct a tree at runtime that reflects the hierarchical nesting structure of individual execution points and constructs.
- Alchemist supports profiling of Read-After-Write (RAW) dependences as well as Write-After-Write (WAW) and Write-After-Read (WAR) ones. RAW dependencies provide a measure of the amount of available concurrency in a program that can be exploited without code transformations, while removing WAR and WAW dependencies typically require source-level changes, such as making private copies of data.
- We evaluate profile quality on a set of programs that have been parallelized elsewhere. We compare the program points highly ranked by Alchemist with those actually parallelized, and observe strong correlation between the two. Using Alchemist, we also manually parallelize a set of benchmarks to quantify its benefits. Our experience shows that, with the help of Alchemist, parallelizing medium-size C programs (on the order of 10K lines of code) can lead to notable runtime improvement on multicore platforms.

II. Overview

Our profiling technique detects code structures that can be run asynchronously within their dynamic context. More precisely, as illustrated in Fig. 1, it identifies code structures like C, delimited by C_{entry} and C_{exit}, which can be spawned as a thread and run simultaneously with C's continuation, the execution following the completion of C. C can be a procedure, loop, or an `if-then-else` construct. Our execution model thus follows the parallelization strategy available using *futures* [10], [15], [25] that has been used to introduce asynchronous execution into sequential Java, Scheme and Lisp programs; it is also similar to the behavior-oriented execution model [7] proposed for C. A future *joins* with its continuation at a claim point, the point at which the return value of the future is needed.

Our goal is to automatically identify constructs that are amenable for *future* annotation and provide direct guidance for the parallelization transformation. The basic idea is to profile the duration of a construct and the time intervals of the two conflicting memory references involved in any dependence from inside the construct to its continuation. Consider the sequential run in Fig. 1. Assume our profiling mechanism reveals a dependence between execution points x and y. The duration of construct C and the interval between x and y are profiled as T_{dur} and T_{dep}, respectively. Let the timestamps of an execution point s in the sequential and the parallel executions be $t_{seq}(s)$ and $t_{par}(s)$, respectively; the interval between x and y in the *parallel* run is then

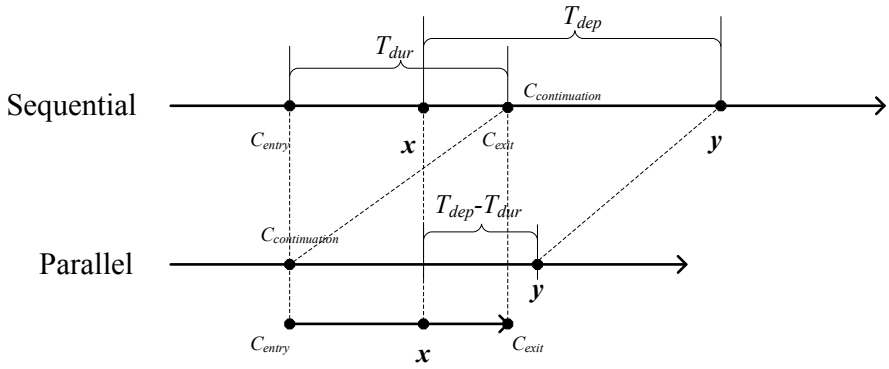


Figure 1. Overview (the dashed lines represent the correspondence between execution points).

```
                                                    Source Position
1.  int zip (in, out)                                   8471
2.  {                                                   8472
3.     while (/* input buffer not empty*/) {            1600
4.        /* process one literal at a time*/            …
5.        if (/*processing buffer full*/)               1662
6.           flush_block (&window[],…);                 1662
7.        flag_buf[last_flags++]=… ;                    6629
8.     }                                                1701
9.     … = flush_block (&window[],…);                   1704
10.    outbuf[outcnt++]= /*checksum*/                   8527
   }
```

Profile

```
1. Method  main              Tdur=20432742,   inst=1
2. Loop  (main,3404)         Tdur=20431181,   inst=1
   … …
9. Method flush_block        Tdur=643408,     inst=2
        RAW:  line 29  → line 9     Tdep=1
        RAW:  line 28  → line 10    Tdep=3
        RAW:  line 14  → line 14    Tdep=4541215
        RAW:  line 22  → line 19    Tdep=4541231
        … …
```

```
                                                    Source Position
11. off_t flush_block (buf, len, …)                     6496
12. {                                                   6495
13.    flag_buf[last_flags] = /*the current flag*/;     6505
14.    input_len + = /* length of the block*/;          6528
15.    /* Encode literals to bits*/
16.    do {                                             6670
17.       … flag =flag_buf[…];                          6671
18.       if (flag … ) {                                6673
19.          if (bi_valid > …) {                        754
20.             outbuf[outcnt++]=(char) bi_buf…;        756
21.             bi_buf= /* encoding*/;                  757
22.             bi_valid + = … ;                        758
23.          }                                          759
24.       }
25.    } while (/*not the last literal*/);              6698
26.    last_flags=0;                                    6064
27.    /* Write out remaining bits*/                     …
28.    outbuf[outcnt++]=(char) bi_buf…;                 790
29.    return /* # of compressed literals*/;            6597
   }
```

Figure 2. Profiling `gzip-1.3.5`.

$$
\begin{aligned}
& t_{par}(y) - t_{par}(x) \\
= \;& (t_{seq}(y) - (t_{seq}(C_{exit}) - t_{seq}(C_{entry}))) - t_{seq}(x) \\
= \;& (t_{seq}(y) - t_{seq}(x)) - (t_{seq}(C_{exit}) - t_{seq}(C_{entry})) \\
= \;& T_{dep} - T_{dur}
\end{aligned}
$$

as shown in Fig. 1.

The profile and the dependence types provide guidance to programmers as follows.

- For RAW dependences, i.e., x is a write and y is a read of the same location, if $T_{dep} > T_{dur}$, construct C is a candidate for asynchronous evaluation with its continuation. That is because it indicates the distance between x and y is positive in the parallel run, meaning it is highly likely that when y is reached, C has already finished the computation at x and thus dependence can be easily respected. A simple parallelization transformation is to annotate C as a future, which is joined at any possible conflicting reads e.g., y. More complex transformations that inject barriers which stall the read at y until it is guaranteed that C will perform no further writes to the same location (e.g., x has completed) are also possible.
- For WAR dependences, i.e., x is a read and y is a write, if C has been decided by the RAW profile to be parallelizable, two possible transformations are suggested to the programmer. The first one is for dependences with $T_{dep} < T_{dur}$, which implies that if the construct is run asynchronously, the write may happen before the read and thus the read may see a value from its logical future, violating an obvious safety property. Therefore, the programmer should create a private copy of the conflict variable in C. For dependences with $T_{dep} > T_{dur}$, the programmer can choose to join the

asynchronous execution before y.

- WAW dependences are similar to WAR.

Consider the example in Fig. 2, which is abstracted from the single file version of `gzip-1.3.5` [1]. It contains two methods `zip` and `flush_block`. To simplify the presentation, we inline methods called by these two procedures and do not show statements irrelevant to our discussion. The positions of the statements in the source that are shown in the code snippet are listed on the right. Procedure `zip` compresses one input file at a time. It contains a `while` loop that processes the input literals, calculating the frequencies of individual substrings and storing literals into temporary buffers. Whenever these buffers reach their limits, it calls procedure `flush_block` to encode the literals and emit the compressed results at line 6. During processing, the `while` loop sets `flags[]` which will be used later during encoding. The procedure `flush_block` first records the current flag and updates the number of processed inputs at lines 13-14; it scans the input buffer within the loop at lines 16-25, encodes a literal into bits stored within buffer `bi_buf`, which is eventually output to buffer `outbuf` at line 20. Variable `bi_valid` maintains the number of result bits in the buffer, and `outcnt` keeps track of the current pointer in the output buffer. After all the literals are processed, the procedure resets the `last_flags` variable at line 26 and emits the remaining bits in the bit buffer. Note that at line 20 in the encoding loop, bits are stored to the output buffer at the unit of bytes and thus at the end, there are often trailing bits remaining. The number of compressed literals is returned at line 29.

The code in Fig. 2 is significantly simplified from the actual program, which is comprised of much more complex control flow and data flow, both further confounded by aliasing. It is difficult for traditional static techniques to identify the places where concurrency can be exploited. Running `gzip` with Alchemist produces the results shown in Fig. 2. Each row corresponds to a source code construct. The profile contains T_{dur}, approximated by the number of instructions executed inside the construct and the number of executions of the construct. For example, the first row shows that the *main* function executes roughly 20 million instructions once. The second row shows that the loop headed by line 3404 in the original source executes roughly 20 million instructions once, i.e. there is one iteration of the loop.

Let us focus on the third row, which corresponds to the execution of calls to procedure `flush_block`. From the profile, we see the procedure is called two times – the first call corresponds to the invocation at line 6, and the other at line 9. Some of the profiled RAW dependences between the procedure and its continuation are listed. Note that while one dependence edge can be exercised multiple times, the profile shows the minimal T_{dep} because it bounds the concurrency that one can exploit. We can see that only the first two (out of a total of fifteen) dependences, highlighted by the box, do not satisfy the condition $T_{dep} > T_{dur}$, and thus hinder concurrent execution. Further inspection shows that the first dependence only occurs between the call site at line 9, which is out of the main loop in `zip()`, and the return at 29. In other words, this dependence does not prevent the call at 6 from being spawned as a future. The second dependence is between the write to `outcnt` at 28 and the read at 10. Again, this dependence only occurs when the call is made at line 9. While these dependencies prevent the call to `flush_block` on line 9 from running concurrently with its continuation (the write to `outbuf`), it does not address the calls to `flush_block` made at line 6. Because these calls occur within the loop, safety of their asynchronous execution is predicated on the absence of dependences within concurrent executions of the procedure itself, as well as the absence of dependencies with operations performed by the outer procedure. Snippets of the profile show that the operation performed at line 14 has a dependence with itself separated by an interval comprising roughly 4M instructions. Observe that the duration of the loop itself is only 2M instructions, with the remaining 2M instructions comprised of actions performed within the iteration after the call. Also observe that there is no return value from calls performed within the loop, and thus dependencies induced by such returns found at line 9, are not problematic here.

Method **flush_block** T_{dur}=643408, inst=2

WAW:	line 28	→ line 10	T_{dep}=7
WAR:	line 17	→ line 7	T_{dep}=6702
WAR:	line 26	→ line 7	T_{dep}=6703
WAR:	line 17	→ line 13	T_{dep}=3915860
… …			

Figure 3. WAR and WAW profile.

Besides RAW dependencies, Alchemist also profiles WAR and WAW dependencies. Unlike a RAW dependence which can be broken by blocking execution of the read access until the last write in the future upon which it depends completes, WAR and WAW dependencies typically require manifest code transformations. The WAR and WAW profile for `gzip` is shown in Fig. 3. The interesting dependences, namely those which do not satisfy $T_{dep} > T_{dur}$, are highlighted in the box. The first dependence is between the two writes to `outcnt` at 28 and 10. Note that there are no WAW dependences detected between writes to `outbuf` as they write to disjoint locations. Another way of understanding it is that the conflict is reflected on the buffer index `outcnt` instead of the buffer itself. As before, this dependence does not compromise the potential for executing calls initiated at 6 asynchronously. The second dependence is caused because the read to `flag_buf[]` happens at 17 and the write happens later at 7. While we might be able to

inject barriers between these two operations to prevent a dependence violation, a code transformation that privatizes `flag_buf[]` by copying its values to a local array is a feasible alternative. Similarly, we can satisfy the third dependence by hoisting the reset of `last_flags` from inside `flush_block ()` and put it in the beginning of the continuation, say, between lines 6 and 7. In the meantime, we need to create a local copy of `last_flags` for `flush_block()`. While the decision to inject barriers or perform code transformations demands a certain degree of programmer involvement, Alchemist helps identify program regions where these decisions need to be made, along with supporting evidence to help with the decision-making process. Of course, as with any profiling technique, the completeness of the dependencies identified by Alchemist is a function of the test inputs used to run the profiler.

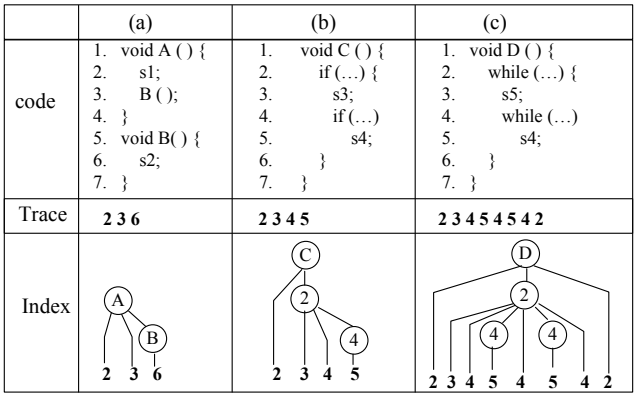


Figure 4. Execution Indexing Examples.

III. The Profiling Algorithm

Most traditional profiling techniques simply aggreate information according to static artifacts such as instructions and functions. Unfortunately, such a strategy is not adequate for dependence profiling. Consider the example trace in Fig. 4 (c). Assume a dependence is detected between the second instance of 5 in the trace and the second instance of 2. Simply recording the time interval between the two instances or increasing a frequency counter is not sufficient to decide available concurrency. We need to know that the dependence indeed crossed the iteration boundaries of loops 2 and 4. This information is relevant to determine if the iterations of these loops are amenable for parallelization. In contrast, it is an intra-construct dependence for procedure `D` and can be ignored if we wish to evaluate calls to `D` concurrently with other actions. Thus, an exercised dependence edge, which is detected between two instructions, has various implications for the profiles of multiple constructs. The online algorithm has to efficiently address this issue.

A. Execution Indexing

RAW, WAR, and WAW dependences are detected between individual instructions at run time. Since our goal is to identify available concurrency between constructs and their futures, intra-construct dependences can be safely discarded. An executed instruction often belongs to multiple constructs at the same time. As a result, a dependence may appear as an intra-construct dependence for some constructs, and as a cross-boundary dependence for others.

In order to efficiently update the profile of multiple constructs upon detection of a dependence, we adopt a technique called *execution indexing* [26] to create an execution index tree that represents the nesting structure of an execution point. Fig. 4 shows three examples of execution indexing. In example (a), node `A` denotes the construct of procedure `A`. As statements 2 and 3 are nested in procedure `A`, they are children of node `A` in the index tree. Procedure `B` is also nested in `A`. Statement 6 is nested in `B`. The *index* for an execution point is the path from the root to the point, which illustrates the nesting structure of the point. For example, the index of the first instance of statement 6 in trace (a) is $[A, B]$. Fig. 4 (b) shows an example for an `if-then-else` construct. The construct led by statement 2 is nested in procedure `C()`, and construct 4 is nested within construct 2, resulting in the index tree in the figure. Note that statement 2 is not a child of node 2, but a child of node `C` because it is considered as being nested in the procedure instead of the construct led by itself. Example (c) shows how to index loops. Since loop iterations are often strong candidates for parallelization, each iteration is considered as an instance of the loop construct so that a dependence between iterations is considered as a cross boundary dependence and hence should be profiled. We can see in the index tree of example (c), the two iterations of loop 4 are siblings nested in the first iteration of 2. The index of 5_2 (the second instance of statement 5) is $[D, 2, 4]$, exactly disclosing its nesting structure. From these examples, we observe that (i) a dynamic instance of a construct is represented by a subtree; (ii) the index for a particular execution point is the path from the root to this point.

While Fig. 4 only shows simple cases, realistic applications may include control structures such as `break`, `continue`, `return`, or even `long_jump`/ `set_jump`; a naive solution based on the syntax of the source code would fail to correctly index these structures. Control flow analysis is required to solve the problem. The intuition is that a construct is started by a predicate and terminated by the immediate post-dominator of the predicate. Similar to calling contexts, constructs never overlap. Therefore, a similar stack structure can be used to maintain the current index of an execution point. More precisely, a push is performed upon the execution of a predicate, indicating the start of a construct. A pop is performed upon the execution

of the immediate post-dominator of the top construct on the stack, marking the end of that construct. The state of the stack is indeed the index for the current execution point.

Rule	Event	Instrumentation
(1)	Enter procedure X	IDS.push(X)
(2)	Exit procedure X	IDS.pop()
(3)	Non-loop predicate at p	IDS.push(p)
(4)	Loop predicate at p	if (p==IDS.top()) IDS.pop(); IDS.push(p)
(5)	Statement s	while (p=IDS.top() $\wedge$ s is the immediate post-dominator of p) IDS.pop()

*IDS is the indexing stack.

Figure 5. Instrumentation Rules for Indexing.

The instrumentation rules for execution indexing are presented in Fig. 5. The first two rules mark the start and end of a procedure construct by pushing and popping the entry associated with the procedure. In rule (3), an entry is pushed if the predicate is not a loop predicate. Otherwise in rule (4), the top entry is popped if the entry corresponds to the loop predicate of the previous iteration, and the current loop predicate is then pushed. Although rule (4) pushes and pops the same value, it is not redundant as these operations have side effects, which will be explained next. By doing so, we avoid introducing a nesting relation between iterations. Finally, if the top stack entry is a predicate and the current statement is the predicate's immediate post-dominator, the top entry is popped (Rule 5). Irregular control flows such as those caused by `long_jump` and `set_jump` are handled in the same way as that presented in [26].

Managing the Index Tree for an Entire Execution. Unfortunately, the above stack-based method that is similar to the one proposed in [26] only generates the index for the current execution point, which is the state of the index stack. It does not explicitly construct the whole tree. However, in Alchemist, we need the tree because a detected dependence may involve a construct that completed earlier. For instance, assume in the execution trace of Fig. 4 (c), a dependence is detected between 5_1 and 2_2. The index of 5_1 is $[D, 2, 4]$. It is nested in the first iteration of loop 4, which has completed before 2_2, and thus its index is no longer maintained by the stack. In order to update the right profiles, 5_1's index needs to be maintained.

A simple solution is to maintain the tree for the entire execution. However, doing so is prohitively expensive and unnecessary. The key observation is that if a construct instance C has ended for a period longer than $T_{dur}(C)$, the duration of the construct instance, it is safe to remove the instance from the index tree. The reason is that any dependence between a point in C and a future point, must satisfy $T_{dep} > T_{dur}(C)$ and hence does not affect the profiling result. The implication is that the index tree can be managed by using a construct pool, which only maintains the construct instances that need to be indexed. Since one node is created for one construct instance regardless of the T_{dur} of the construct, only those constructs that get repeatedly executed have many instances at runtime and pose challenges to index tree management.

Table I
The Algorithm for Managing the Index Tree.

pc is the program counter of the head of the construct.
PROFILE constrains the profile for constructs, indexed by pc.
$pool$ is the construct pool.

```
 1: IDS.push (pc)
 2: {
 3:     c=pool.head();
 4:     while (timestamp − c.T_exit < c.T_exit − c.T_enter) {
 5:         c=pool.next();
 6:     }
 7:     pool.remove(c);
 8:     c.label= pc;
 9:     c.T_enter= timestamp;
10:     c.T_exit= 0;
11:     c.parent= IDS[top-1];
12:     IDS[top++]=c;
13: }
14:
15: IDS.pop ()
16: {
17:     c=IDS[− − top];
18:     c.T_exit=timestamp;
19:     pc=c.label;
20:     PROFILE[pc].T_total+=c.T_exit-c.T_enter;
21:     PROFILE[pc].inst++;
22:     pool.append(c)
23: }
```

Theorem 1: Assume the maximum size of an instance of a repeatedly executed construct is M, a statement can serve as the immediate post-dominator for a maximum of N constructs, and the maximum nesting level is L. The memory requirement of Alchemist is $O(M \cdot N + L)$.

Proof: Let i be the instruction count of an execution point, constructs completed before $i - M$ are of no interest because any dependence between i and any point in those constructs must have $T_{dep} > M$. Thus, only constructs that completed between $i - M$ and i need to be indexed with respect to i, i.e., the nodes for those constructs can not be retired. As the maximum number of constructs that can complete in one execution step is N according to rule (5) in Fig. 5, the number of constructs completed in that duration can not exceed $M \cdot N$. Since L is the maximum number of active constructs, i.e., constructs that have not terminated, the space complexity is $O(M \cdot N + L)$. ■

The theorem says that the memory requirement of Alchemist is bounded if the size of any repeatedly executed construct is bounded. In our experiment, a pre-allocated pool of the size of one million dynamic constructs never led to memory exhaustion. The pseudo-code for the algorithm is presented in Table I. It consists of two functions: `IDS.push` and `IDS.pop`. In the instrumentation rules presented earlier, they are the push and pop operations of the index stack.

In the algorithm, variable $pool$ denotes the construct pool. Upon calling the push operation with the program counter of the head instruction of a construct, usually a predicate or a function entry, the algorithm finds the first available construct from the pool by testing if it satisfies the condition at line 4. Variable $timestamp$ denotes the current time stamp. It is simulated by the number of executed instructions. If the predicate is true, c can not be retired and the next construct from $pool$ is tested. The first construct that can be safely retired is reused to store information for the newly entered construct. Lines 8-11 initialize the construct structure c. Specifically, line 11 establishes the connection from c to its enclosing construct, which is the top construct on the stack. Line 12 pushes c to the stack.

Upon calling the pop function, the top construct is popped. Its ending time stamp is recorded at line 18. The profile of the popped construct, indexed by its pc in the `PROFILE` array, is updated. The total number of executed instructions for the construct is incremented by the duration of the completed instance. Note that a construct may be executed multiple times during execution. The number of executed instances of the construct is incremented by one. Finally, the data structure assigned to the completed construct instance is appended to the construct pool so that it might be reused later on. We adopt a lazy retiring strategy – a newly completed construct is attached to the tail of the pool while reuse is tried from the head. Hence, the time a completed construct remains accessible is maximized.

B. The Profiling Algorithm

Function `profile()` in Table II explains the profiling procedure. The algorithm takes as input a dependence edge denoted as a tuple of six elements. The basic rule is to update the profile of each nesting construct bottom up from the enclosing construct of the dependence head up to the first active (not yet completed) construct along the head's index. This is reflected in lines 7 and 14. The condition at 7 dictates that a nesting construct, if subject to update, must have completed ($c.T_{enter} < c.T_{exit}$[1]) and must not retire. If a construct has not completed, the dependence must be an intra-dependence for this construct and its nesting ancestors. If the construct has retired and its residence memory space c has been reused, it must be true that T_h falls out of the duration of the current construct occupying c and thus condition at 7 is not true. Lines 8-13 are devoted to updating the profile. It first tests if the dependence has been recorded. If not, it is simply added to the construct's profile. If so, a further test to determine if the T_{dep} of the detected dependence is smaller than the recorded minimum T_{dep} is performed. If yes, the minimum T_{dep} is updated.

To illustrate the profiling algorithm, consider the example trace and its index in Fig. 4 (c). Assume a dependence is

[1] T_{exit} is reset upon entering a construct.

Table II
Profiling Algorithm.

pc_h and pc_t are the program counters for the head and tail of the dependence.
c_h, c_t are the construct instances in which the head and tail reside.
T_h, T_t are the timestamps.

```
1: Profile(pc_h, c_h, T_h, pc_t, c_t, T_t)
2: {
3:     T_dep=T_t-T_h;
4:     pc=c_h.label;
5:     P=PROFILE[pc];
6:     c=c_h;
7:     while (c.T_enter <= T_h < c.T_exit) {
8:         if (P.hasEdge (pc_h → pc_t) {
9:             T_min=P.getTdep (pc_h → pc_t);
10:            if (T_min > T_dep)
11:                P.setTdep (pc_h → pc_t, T_dep);
12:        } else
13:            P.addEdge (pc_h → pc_t, T_dep);
14:        c=c.parent;
15:    }
16: }
```

detected between 5_2, with index $[D, 2, 4]$, and 2_2, with index $[D]$. The input to `profile` is the tuple $< pc_h = 5,\ c_h = \widehat{4^r},\ T_h = 6,\ pc_t = 2,\ c_t = \widehat{D},\ T_t = 8 >$, in which $\widehat{n}$ represents a construct headed by n. $\widehat{4^r}$ represents the node 4 on the right. The algorithm starts from the enclosing construct of the head, which is $\widehat{4^r}$. As $T_{enter}(\widehat{4^r}) = 6$ and $T_{exit}(\widehat{4^r}) = 7$, the condition at line 7 in Table II is satisfied as $T_h = 6$; the profile is thus updated by adding the edge to `PROFILE[4]`. The algorithm traverses one level up and looks at $\widehat{4^r}$'s parent $\widehat{2}$. The condition is satisfied again as $T_{enter}(\widehat{2}) = 2 < T_h = 6 < T_{exit}(\widehat{2}) = 8$; thus, the dependence is added to `PROFILE[2]`, indicating that it is also an external dependence for $\widehat{2}$, which is the first iteration of the outer while loop in code Fig 4(c). However, the parent construct $\widehat{D}$ is still active with $T_{exit} = 0$; thus, the condition is not satisfied here and `PROFILE[1]` is not updated.

Recursion. The algorithm in Table I produces incorrect information in the presence of recursion. The problem resides at line 20, where the total number of executed instructions of the construct is updated. Assume a function `f()` calls itself and results in the index path of $[f_1, f_2]$. Here we use the subscripts to distinguish the two construct instances. Upon the end of $\widehat{f_1}$ and $\widehat{f_2}$, $T_{dur}(\widehat{f_1})$ and $T_{dur}(\widehat{f_2})$ are aggregated to PROFILE[f].T_{total}. However, as $\widehat{f_2}$ is nested in $\widehat{f_1}$, the value of $T_{dur}(\widehat{f_2})$ has already been aggregated to $T_{dur}(\widehat{f_1})$, and thus is mistakenly added twice to the T_{total}. The solution is to use a nesting counter for each pc so that the profile is aggregated only when the counter reaches zero.

Inadequacy of Context Sensitivity. In some of the recent work [6], [8], context sensitive profiling [2] is used to collect dependence information for parallelization. However,

context sensitivity is not sufficient in general. Consider the following code snippet.

```
F() {
  for (i...)
    for (j...)
      A();
      B();
    }
  }
}
```

Assume there are four dependences between some execution inside `A()` and some execution inside `B()`. The first one is within the same `j` iteration; the second one crosses the `j` loop but is within the same `i` iteration; the third one crosses the `i` loop but is within the same invocation to `F()`; the fourth one crosses different calls to `F()`. They have different implications for parallelization. For instance, in case one, the `j` loop can be parallelized; in case two, the `i` loop can be parallelized but the `j` loop may not; and so on. In all the four cases, the calling context is the same. In case four, even using a loop iteration vector [6] does not help.

IV. Experimentation

Alchemist is implemented on valgrind-2.2.0 [19]. The evaluation consists of various sequential benchmarks. Some of them have been considered in previous work and others (to the best of our knowledge) have not. Relevant details about the benchmarks are shown in Table III and discussed below.

Table III
Benchmarks, number of static/dynamic constructs and running times given in seconds.

Benchmark	LOC	Static	Dynamic	Orig.	Prof.
197.parser	11K	603	31,763,541	1.22	279.5
bzip2	7K	157	134,832	1.39	990.8
gzip-1.3.5	8K	100	570,897	1.06	280.4
130.li	15K	190	13,772,859	0.12	28.8
ogg	58K	466	4,173,029	0.30	70.7
aes	1K	11	2850	0.001	0.396
par2	13K	125	4437	1.95	324.0
delaunay	2K	111	14,307,332	0.81	266.3

A. Runtime

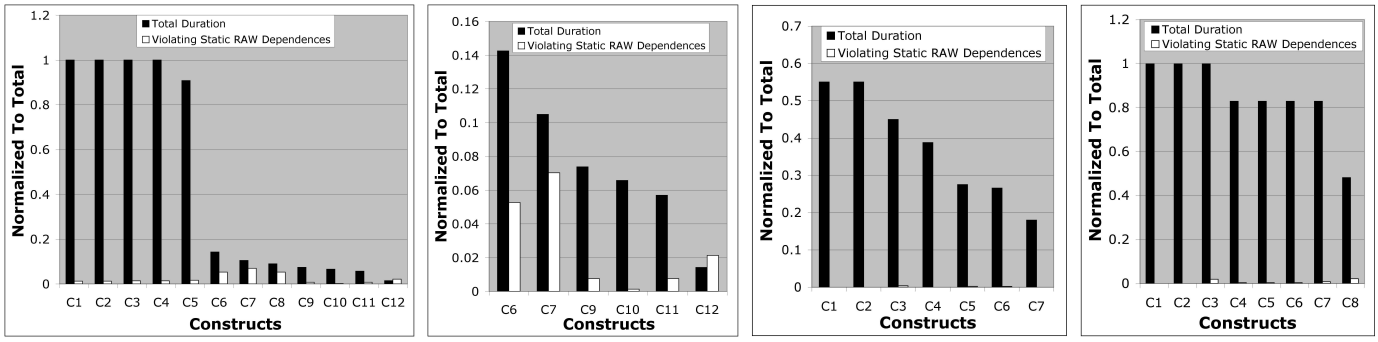
The first experiment is to collect the runtime overhead of Alchemist. The performance data is collected on a Pentium Dual Core 3.2 GHZ machine with 2GB RAM equipped with Linux Gentoo 3.4.5. The results are presented in Table III. Columns `Static` and `Dynamic` present the number of static and dynamic constructs profiled. Column `Orig.` presents the raw execution time. Column `Prof.` presents the times for running the programs in Alchemist. The slow down factor ranges from 166-712 due to dependence detection and indexing. Note that the valgrind infrastructure itself incurs 5-10 times slowdown. The numbers of static unique constructs and their dynamic instances are presented in the third column of Table III. According to the profiling algorithm mentioned earlier, profile is collected per dynamic construct instance and then aggregated when the construct instance is retired. As mentioned earlier, we used a fix-sized construct pool so that the memory overhead is bounded. The pool size is one million, with each construct entry in the pool taking 132 bytes. We have not encountered overflow with such a setting. Since Alchemist is intended to be used as an offline tool, we believe this overhead is acceptable. Using a better infrastructure such as Pin [18] may improve runtime performance by a factor of 5-8, and implementing the optimizations for indexing as described in [26] may lead to another 2-3 factor improvement.

B. Profile Quality

The next set of experiments are devoted to evaluating profile quality. To measure the quality of the profiles generated by Alchemist we run two sets of experiments. For the first set of experiments, we consider sequential programs that have been parallelized in previous work. We observe how parallelization is reflected in the profile. We also evaluate the effectiveness of Alchemist in guiding the parallelization process by observing the dependences demonstrated by the profile and relating them to the code transformations that were required in the parallelization of the sequential programs. Furthermore, we run Alchemist on a sequential program that can not be migrated to a parallel version to see if the profile successfully shows that the program is not amenable for parallelization.

For our other set of experiments, we parallelize various programs using the output from Alchemist. We first run the sequential version of the program through Alchemist to collect profiles. We then look for large constructs with few violating static RAW dependences and try to parallelize those constructs. To do so, we use the WAW and WAR profiles as hints for where to insert variable privatization and thread synchronization between concurrently executing constructs in the parallel implementation. We assume no specific runtime support and parallelize programs using POSIX threads (pthreads).

1) Parallelized Programs: We first consider programs parallelized in previous work. We claim that a given construct C is amenable for asynchronous evaluation if 1) the construct is large enough to benefit from concurrent execution and 2) the interval between its RAW dependences are greater than the duration of C. To verify our hypothesis we examined programs *gzip, 197.parser* and *130.lisp* parallelized in [7]. The programs were parallelized by marking possibly parallel regions in the code and then



(a) Gzip Profile 1 (b) Gzip Profile 2 (c) 197.parser Profile (d) 130.lisp Profile

Figure 6. Size and number of violating static RAW dependences for constructs parallelized in [7]. For Figures 6(a) and 6(b), parallelized constructs $C1$ and $C9$ represent a loop on line 3404 and `flush_block`, respectively. For *gzip*, Fig. 6(b) shows the constructs that remain after $C1$ and all nested constructs with a single instance per instance of $C1$ have been removed. Construct $C3$ in Fig. 6(c) represents the parallelized loop on line 1302 in *197.parser* and construct $C2$ in Fig. 6(d) represents the parallelized loop on line 63 in *130.lisp*.

running in a runtime system that executes the marked regions speculatively.

Fig. 6 shows the profile information collected by Alchemist for the sequential versions of *gzip, 197.parser* and *130.lisp*. The figure shows the number of instructions (i.e. total duration) and the number of violating static RAW dependences for the constructs that took the most time in each program. The duration of the constructs are normalized to the total number of instructions executed by the program, and the number of violating static RAW dependences are normalized to the total number of violating static RAW dependences detected in the profiled execution. Intuitively, a construct is a good candidate if it has many instructions and few violating dependences.

Gzip v1.3.5. The loop on line 3404 and the `flush_block` procedure were parallelized in [7]. In Figures 6(a) and 6(b) construct $C1$ represents the loop and $C9$ represents `flush_block`. The figure clearly shows that $C1$ is a good candidate for concurrent execution because it is the largest construct and has very few violating RAW depedences. Parallelizing $C1$ makes constructs $C2, C3, C4, C5$, and $C8$ no longer amenable to parallelization because the constructs only have a single nested instance for each instance of $C1$. In other words, the constructs are parallelized too as a result of $C1$ being parallelized. Thus, to identify more constructs amenable for asynchronous execution, we removed constructs $C1, C2, C3, C4, C5$, and $C8$. The remaining constructs are shown in Fig. 6(b). Constructs $C9, C10$ and $C11$ have the fewest violations out of the remaining constructs, and the largest construct $C9$ (`flush_block`) becomes the next parallelization candidate. The Alchemist WAW/WAR profile pinpointed the conflicts between `unsigned short out_buf` and `int outcnt` and `unsigned short bi_buf` and `int bi_valid` that were mentioned in [7].

197.parser. *Parser* has also been parallelized in [7]. Fig. 6(c) presents the profile information. Construct $C3$ corresponds to the loop (on line 1302) which was parallelized. Inspection of the code revealed that constructs $C1$ and $C2$ (corresponding respectively to the loop in `read_dictionary` and method `read_entry`), while both larger than $C3$ and with less violating dependences, were unable to be parallelized because they were I/O bound.

130.lisp. *XLisp* from Spec95, also parallelized by [7], is a small implementation of lisp with object-oriented programming. It contains two control loops. One reads expressions from the terminal and the other performs batch processing on files. In the parallel version, they marked the batch loop as a potentially parallel construct to run speculatively in their runtime system. Construct $C2$ in Fig.6(d) corresponds to the batch loop in `main`. $C1$ corresponds to method `xlload` which is called once before the batch loop, and then a single time for each iteration of the loop. The reason $C1$ executed slightly more instructions than $C2$ was because of the initial call before the loop. Thus parallelizing construct $C2$, as was done in the previous work, results in all but one of the calls to `xlload` to be executed in parallel.

Delaunay Mesh Refinement. It is known that parallelizing the sequential Delaunay mesh refinement algorithm is extremely hard [16]. The result of Alchemist provides confirmation. In particular, most computation intensive constructs have more than 100 static violating RAW dependences. In fact, the construct with the largest number of executed instructions has 720 RAW dependences.

2) Parallelization Experience: For the following benchmarks, we have used profiles generated by Alchemist to implement parallel versions of sequential programs. We report our experience using Alchemist to identify opportunities for parallelization and to provide guidance in the parallelization process. We also report speedups achieved by the parallel version on 2 dual-core 1.8GHZ AMD Opteron(tm) 865 processors with 32GB of RAM running Linux kernel version

2.6.9-34.

Table IV
PARALLELIZATION EXPERIENCE:THE PLACES THAT WE PARALLELIZED AND THEIR PROFILES.

Progam	Code Location	Static Conflict		
		RAW	WAW	WAR
bzip2	6932 in main()	3	103	0
	5340 in compressStream()	23	53	63
ogg	802 in main()	6	30	17
aes	855 in AES_ctr128_encrypt	0	7	3
par2 par2	887 in Par2Creator:: ProcessData ()	1	12	19
	489 in Par2Creator:: OpenSourceFiles()	0	2	12

Par2cmdline. *Par2cmdline* is a utility to create and repair data files using Reed Solomon coding. The original program was 13718 lines of C++ code. We generated a profile in Alchemist by running *par2* to create an archive for four text files. By looking at the profile we were able to parallelize the loop at line 489 in `Par2Creator::OpenSourceFiles` and the loop at line 887 in `Par2Creator::ProcessData`. The loop at line 489 was the second largest construct and only had one violating static RAW dependence. The Alchemist profile detected a conflict when a file is closed. The parallel version moved file closing to guarantee all threads are complete before closing files. The loop at line 887 was the eighth largest construct with no violating static RAW dependences and thus is the second most beneficial place to perform parallelization. The loop processes each output block. We parallelized this loop by evenly distributing the processing of the output blocks among threads. We ran the parallel version and the sequential version on the same large 42.5MB WAV file to create an archive. The parallel version took 6.33 seconds compared to the sequential version which completed in 11.25 seconds (speedup of 1.78).

Bzip2 v1.0. *Bzip2* takes one or more files and compresses each file separately. We ran *bzip2* in Alchemist on two small text files to generate profiling information. With the guidance of the Alchemist profile, we were able to write a parallel version of *bzip2* that achieves near linear speedup. The first construct we were able to parallelize was a loop in `main` that iterates over the files to be compressed. This was the single largest construct in the profile and had only 3 violating dependences. The WAW dependences shown in the profile indicate a naive parallelization would conflict on the shared `BZFILE *bzf` structure and the data structures reachable from `bzf` such as `stream`. In the sequential program, this global data structure is used to keep track of the file handle, current input buffer and output stream. When parallelizing *bzip2* to compress multiple files concurrently each thread has a thread local `BZFILE` structure to operate on.

After parallelizing the first construct we removed it from the profile along with all nested constructs with only a single instance per iteration of the loop, as we did in Fig. 6(b) with *gzip*. From the new constructs Alchemist identified the opportunity to compress multiple blocks of a single file in parallel, although the construct had an unusually high number of violating static RAW dependences. Further inspection showed the RAW dependences identified by the profile resulted from a call to `BZ2_bzWriteClose64` after the loop. Each iteration of the loop compresses 5000 byte blocks and if there is any data that is left over after the last 5000 byte block, `BZ2_bzWriteClose64` processes that data and flushes the output file. As with the loop in `main`, the profile reported many WAW and WAR dependences on the `bzf` structure. By examining the violating dependences reported by Alchemist, we were able to rewrite the sequential program so that multiple blocks could be compressed in parallel. The parallelization process included privatizing parts of the data in the `bzf` structure to avoid the reported conflicts. The parallel version of *bzip2* achieves near-linear speedup both for compressing multiple files and compressing a single file. We compressed two 42.5MB wav files with the original sequential *bzip2* and our parallel version with 4 threads. The sequential version took 40.92 seconds and the parallel version took 11.82 seconds resulting in a speedup of 3.46.

AES Encryption (Counter Mode) in OpenSSL. AES is a block cipher algorithm which can be used in many different modes. We extracted the AES counter mode implementation from *OpenSSL*. Based on the profile generated by Alchemist while encrypting a message of size 512 (32 blocks each 128 bits long) we parallelized the implementation. The encryption algorithm loops over the input until it has read in an entire block and then calls `AES_encrypt` to encrypt the block and then makes a call to `AES_ctr128_inc(ivec)` to increment the `ivec`. The incremented `ivec` is then used by the next call to `AES_encrypt` to encrypt the next block. We parallelized the main loop that iterates over the input (sixth largest construct), which had no violating static RAW dependences in the profile. The WAW/WAR dependences in the profile included conflicts on `ivec`. In our parallel version each thread has its own `ivec` and must compute its value before starting encryption.

Table V
PARALLELIZATION RESULTS.

Benchmark	Seq.(sec.)	Par. (sec.)	Speedup
bzip2	40.92	11.82	3.46
ogg	136.27	34.46	3.95
par2	11.25	6.33	1.78
aes_ctr	9.46	5.81	1.63

To evaluate the performance of our parallel encryption implementation we encrypted a message that had 10 million blocks. Each block in AES is 128 bits. We executed on 4

threads. The sequential encryption took 9.46 seconds and the parallel encryption took 5.81 seconds resulting in a speedup of 1.63.

Oggenc-1.0.1. *Oggenc* is a command-line encoding tool for Ogg Vorbis, a lossy audio compression scheme. The version we use contains 58417 lines of code. We ran *oggenc* in Alchemist on two small WAV files each about 16KB and used the profile to parallelize the program. The largest construct identifies the main loop that iterates over the two files being encoded. The profile identifies 6 violating static RAW dependences. Among the detected violations for the loop is a dependence on the `errors` flag that identifies if an error occurred in encoding any of the files. There are also conflicts on a variable used to keep track of the samples read. We parallelized the loop with POSIX threads so that multiple files can be encoded in parallel. Each thread has a local `errors` flag that keeps track of whether or not an error occurred in the encoding of its file. Every thread also has a local count of samples read. Our parallel version of *oggenc* takes as long as the most time consuming file. Our parallel version uses 4 threads. We used the parallel version to encode 4 large WAV files. As expected the parallel version achieved nearly linear speedup (3.95) with the sequential version taking 136.27 seconds and the parallel version taking 34.46 seconds.

V. Related Work

Profiling For Concurrency. Many existing program parallelization projects have profiling as an essential component. This is because statically identifying available parallelism, especially data parallelism, is hard. TEST[5] is a hardware profiler for decomposing java programs into speculative threads. It focuses on decompositions formed from loops. It profiles the minimum dependence distance between loop iterations and uses that to guide thread-level speculation (TLS). Compared to TEST, Alchemist is a software profiler that targets C programs and is more general since it does not rely on TLS to remedy WAR and WAW dependencies at runtime, but rather provides feedback to the programmer who can then perform necessary code transformation to resolve these dependencies, and it treats any program construct as a potential parallelization candidate. POSH [17] is another TLS compiler that exploits program structure. It considers loop iterations and their continuations, subroutines and their continuations as the candidates for speculative thread composition. It models the effects of concurrent execution, thread squashing, and prefetching that closely simulate program execution under TLS. POSH is capable of profiling various constructs, which is similar to Alchemist. However, as the profiling algorithm is not explained in detail, it is not clear how the authors handle the problem of updating profiles for nesting constructs upon detecting dependences. Furthermore, like TEST, WAW and WAR dependences are not profiled. Behavior oriented parallelization [7] is a technique that supports speculatively executing regions of sequential programs. It includes a profiler that detects dependences between execution phases. A transaction-based mechanism is employed to support speculation. In [23], Praun *et al.* analyze parallelization opportunities by quantifying dependences among critical sections as a density computed over the number of executed instructions. Their work collects profile for critical regions guarded by synchronization or constructs annotated by programmers as candidates for speculative execution. Notably, their task level density profile does not provide direct guidance for programmers. ParaMeter [20] is an interactive program analysis and visualization system for large traces to facilitate parallelization. It features fast trace compression and visualization by using BDD. In comparison, Alchemist is a profiler that does not record the whole trace. It would be interesting to see if indexing can be integrated with ParaMeter to present a hierarchical view of traces. Recently, context sensitive dependence profiling is used for speculative optimization and parallelization in [6], [8]. However context sensitivity is not adequate to model loop carry dependences. In comparison, the novelty of Alchemist lies in using index trees to provide more fine-grained information.

Program Parallelization. Traditional automatic program parallelization exploits concurrency across loop iterations using array dependence analyses [3], [9]. In programs which exhibit more complex dataflow and control-flow mechanisms, these techniques are not likely to be as effective. Parallelizing general sequential programs in the presence of side-effects has been explored in the context of the Jade parallel programming language [21]. A Jade programmer is responsible for delimiting code fragments (tasks) that could be executed concurrently and explicitly specifying invariants describing how different tasks access shared data. The runtime system is then responsible for exploiting available concurrency and verifying data access invariants in order to preserve the semantics of the serial program. Recently, speculative parallel execution was shown to be achievable through thread level speculation [17], [14], [22], [28]. These techniques speculatively execute concurrent threads and revoke execution in the presence of conflicts. Software transactional memory (STM) [13], [11], [12] ensures serializable execution of concurrent threads. Kulkarni *et. al* [16] present their experience in parallelizing two large-scale irregular applications using speculative parallelization. Bridges *et. al* [4] show that by using a combination of compiler and architectural techniques, one can effectively parallelize a large pool of SPEC benchmarks.

Our work is also related to context-sensitive profiling [2], in which simple performance measurements are associated with calling contexts. Alchemist demands more fine-grained context information. Central to Alchemist's design is the challenging issue of managing index trees. The unique charateristics of dependence, e.g., it is a two-tuple relation,

also distinguishes our design. The concept of *forward control dependence graph* (FCDG) proposed by Sarkar [29] for the purpose of parallelization is similar to our indexing tree. The difference is that indexing tree is dynamic and FCDG is static.

VI. Conclusion

This paper describes Alchemist, a novel dependence profiling infrastructure. Alchemist is *transparent*, and treats any program construct as a potential candidate for parallelization. Dependence profile for all constructs can be collected by one run using execution indexing technique. Alchemist is *general*, as it does not rely on any specialized hardware or software system support. We argue Alchemist is *useful* because it provides direct guidance for programmers to perform parallelization transformation. Experimental results show that Alchemist provides high quality information to facilitate parallelization with acceptable profiling overhead.

References

[1] *http://people.csail.mit.edu/smcc/projects/single-file-programs/*.

[2] G. Ammons and T. Ball and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI'97*, pages 85–96.

[3] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4), pages 345–420, 1994.

[4] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *MICRO '07*, pages 69–84.

[5] M. Chen and K. Olukotun. Test: A tracer for extracting speculative threads. In *CGO '03*, pages 301–312.

[6] T. Chen, J. Lin, X. Dai, W.C. Hsu, and P.C. Yew. Data Dependence Profiling for Speculative Optimization. In *CC'04*, pages 57–72.

[7] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software Behavior Oriented Parallelization. In *PLDI'07*, pages 223–234, San Diego, CA.

[8] K.F. Faxen, K. Popov, L. Albertsson, S. Janson. Embla - Data Dependence Profiling for Parallel Programming In CISIS'08, pages 780–785.

[9] M. W. Hall, B. R. Murphy, S. P. Amarasinghe, S. Liao, and M. S. Lam. Interprocedural Analysis for Parallelization. In *LCPC'06*, pages 61–80.

[10] R. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *TOPLAS*, 7(4):pages 501–538, 1985.

[11] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *OOPSLA'03*, pages 388–402.

[12] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software Transactional Memory for Dynamic-Sized Data Structures. In *PODC'03*, pages 92–101.

[13] M. Herlihy and E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA'93*, pages 289–300.

[14] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut Program Decomposition for Thread-Level Speculation. In *PLDI'04*, pages 59–70.

[15] R. Kelsey, W. Clinger, and J. Rees. Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(3):pages 7–105, 1998.

[16] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic Parallelism Requires Abstractions. In *PLDI'07*, pages 211–222.

[17] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: A TLS Compiler that Exploits Program Structure. In *PPOPP'06*, pages 158–167.

[18] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI'05*, pages 190-200.

[19] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI '07*, pages 89–100.

[20] G. D. Price, J. Giacomoni, and M. Vachharajani. Visualizing Potential Parallelism in Sequential Programs. In *PACT '08*, pages 82–90.

[21] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A High-Level, Machine-Independent Language for Parallel Programming. *IEEE Computer*, 26(6):pages 28–38, 1993.

[22] J. Greggory Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-Level Speculation. In *ISCA'00*, pages 1–12.

[23] C. v. Praun, R. Bordawekar, and C. Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *PPoPP '08*, pages 185–196.

[24] C. v. Praun, L. Ceze, and C. Cascaval. Implicit parallelism with ordered transactions. In *PPOPP'07*, pages 79–89.

[25] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *OOPSLA'05*, pages 439–453.

[26] B. Xin, N. Sumner, and X. Zhang. Efficient program execution indexing. In *PLDI '08*, pages 238–248.

[27] J. Hardwick. Implementation and Evaluation of an Efficient Parallel Delaunay Triangulation Algorithm. In *SPAA'97*, pages 23–25.

[28] L. Rauchwerger Y. Zhan and J. Torrellas. Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors. In *HPCA'98*, pages 162–173.

[29] V. Sarkar. Automatic partitioning of a program dependence graph into parallel tasks. In IBM Journal of Research and Development, pages 779–804, 1991.