Program Demultiplexing: Data-flow based Speculative Parallelization of Methods in Sequential Programs

Saisanthosh Balakrishnan and Gurindar S. Sohi
Computer Sciences Department, University of Wisconsin-Madison
{sai, sohi}@cs.wisc.edu

Abstract

We present Program Demultiplexing (PD), an execution paradigm that creates concurrency in sequential programs by "demultiplexing" methods (functions or subroutines). Call sites of a demultiplexed method in the program are associated with handlers that allow the method to be separated from the sequential program and executed on an auxiliary processor. The demultiplexed execution of a method (and its handler) is speculative and occurs when the inputs of the method are (speculatively) available, which is typically far in advance of when the method is actually called in the sequential execution. A trigger, composed of predicates that are based on program counters and memory write addresses, launches the speculative execution of the method on another processor.

Our implementation of PD is based on a full-system execution-based chip multi-processor simulator with software to generate triggers and handlers from an x86-program binary. We evaluate eight integer benchmarks from the SPEC2000 suite —programs written in C with no explicit concurrency and/or motivation to create concurrency— and achieve a harmonic mean speedup of 1.8x with our implementation of PD.

1. Introduction

Chip makers are turning to multicore systems as a way to extend Moore's law. While applications such as web servers, database servers, and scientific programs that have abundant software threads will immediately benefit from the many cores, novel approaches are required for executing traditionally-sequential applications in parallel. A key for this will be an understanding of the nature and characteristics of the computations that need to be performed in the application, and the constraints introduced when such computations are expressed in the traditional manner as a totally ordered sequence of instructions. Instruction-level parallel processors already try to unravel some parallelism from this expressed sequential order from a window of in-flight instructions. Speculative parallelization techniques [1, 8, 9, 17, 25, 38, 40, 43] try to overcome the limitations of traditional parallelization techniques by creating threads usually consisting of loop iterations, method continuations, or more generic tasks, speculatively executing many such threads concurrently, and with additional hardware support to detect violations of data and/or control dependencies.

The key limitation of such approaches is the difficulty in reaching "distant" parallelism in a program since the *instantiation* of threads for speculative execution is based on the control-flow sequence of the program. The *commit* ordering of threads is also usually defined (or tightly coupled) by this sequence. Therefore, reaching a distant thread in a program often requires all prior threads in the execution sequence to be identified (by prediction or execution) and scheduled for execution. Data or control-dependence violation(s) in one of these threads may lead to the squashing of all subsequent threads (or as an optimization, threads that violated data and control dependencies) hence wasting execution resources.

This paper proposes a speculative data-flow approach to the problem of parallelizing a sequential application, especially in an environment where the basic processing node is comprised of multiple processors (as in multicore systems). In modern programming languages, a desired sub-computation is expressed as a *method* (also commonly referred to as a function, procedure, or a sub-routine). Observing that a sequential program is a collection of different methods that have been interleaved, or *multiplexed*, both for convenience in expressing the computation and to satisfy the default assumption of execution on a single processor, we propose *Program Demultiplexing*. In Program Demultiplexing (PD), different methods are "demultiplexed" from the sequential order, decoupling the execution of a method from where it is called in the program (the call site). In sequential execution, the call site of a method also represents the beginning of execution of that method, while the execution of a method in PD occurs well before that on an auxiliary processor, albeit speculatively. This execution usually happens after the method is *ready*, – i.e., after its data dependencies are satisfied for that execution instance. Its results are committed if they are valid, when the call site is later reached by the program running on the main processor.

With PD we attempt to achieve data-flow style parallel (speculative) execution, on multiple processors, for an application written in an imperative language for execution on a single processor. Figure 1 illustrates the basic idea of PD.

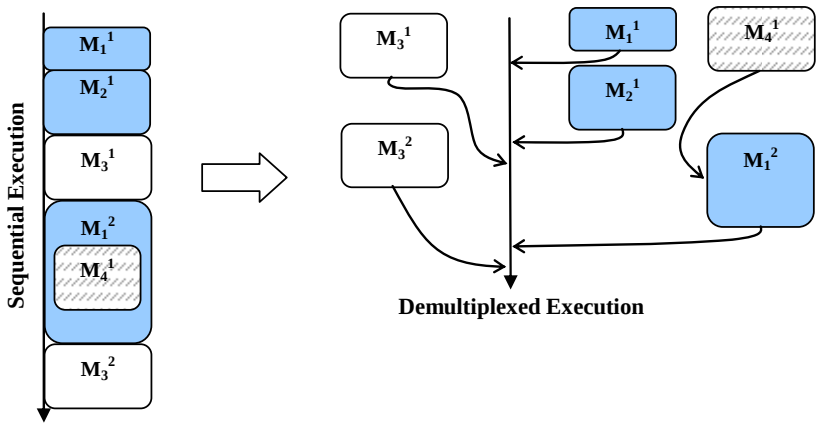


Figure 1. Program Demultiplexing. The left portion of the figure illustrates the sequential execution of a program. The boxes are labeled with names of methods and their execution instances in superscript. Methods with the same shading represent data dependent methods and hence have to be executed serially. The timeline of execution represents runtime. The right portion of the figure illustrates PD execution of the same program. The PD timeline indicates the program using (and thereby, committing) the execution results of demultiplexed methods finishes faster.

Sequential execution corresponds to a total ordering of the methods of the application (shown on the left). In the figure, methods with like shading have data dependences; methods with different shading are independent. The sequential total ordering creates additional (false) control dependences. PD demultiplexes the methods from the total sequential order, creating a partial order, and allows multiple chains of the partial order to be executed in parallel on multiple processors. The demultiplexed methods begin speculative execution earlier than when they are called in sequential execution, but rarely violate data dependencies (due to the total order), allowing a method's execution to be overlapped with the execution of other methods in the program.

The novel aspect of PD is the unordered demultiplexed execution of distant parts of the program when they are ready, i.e., according to data-flow, and not according to control-flow, as is the case with other speculative parallelization models. Nevertheless, previous models are orthogonal to PD, and can be used to further parallelize demultiplexed execution of a method. We choose speculative execution at the granularity of methods for two reasons. First, it allows programmers to easily reason about unstructured parallelism that PD exploits and to possibly create and control concurrency by providing hints and pragmas to the runtime system and/or compiler (unlike explicit multi-threaded programming). Second, with wider adoption of object oriented programming languages, future (and even current) generation of programs are likely to be collections of entities that are more structured and decomposed into objects and libraries, each accessing only a well-defined subset of the global program state, introducing more opportunities for PD.

The rest of this paper is organized as follows: in Section 2, we discuss the concept of Program Demultiplexing and illustrate examples of demultiplexing methods from some SPEC CPU2000 integer benchmarks. In Sections 3 and 4, we present our implementation of PD and initial evaluation results, respectively. We discuss related work in Section 5 and conclude in Section 6.

2. Non-Sequential Execution of Methods

2.1 Methods

A method, also referred to as a function, procedure, or a subroutine, is a sequence of one or more blocks of program statements that performs a component task of a larger program. The same method can be called from different parts of the program and the location from where it is called by the *callee* is the *call site*. When called (perhaps, using one or more parameters), the method carries out some computation, and may return a value back to the caller, which continues execution with the returned value. In this paper, we assume an Intel x86-type architecture that communicates all parameters via the stack. Compiler optimizations (and other architectures) that use registers for a limited set of parameters can also be easily accommodated with minor changes.

Computation in a method can access and modify two types of program state: *local* and *global*. The local state is the state that is not visible outside the method[1]. In a sequential implementation, this state would typically be implemented using a stack. The global state is visible to other program entities outside the method. The *read set* and the *write set* of a method's execution are the set(s) of global state (memory addresses) that the method reads or modifies, respectively. The nature of these sets is defined by the programming language. Most procedural languages

[1] There are a few exceptions to this, such as when parameters are passed by reference, in which the callee's state is accessed by the caller.

usually allow methods to access and make changes to the entire program state. Therefore, after the execution of a method, its changes could be visible to the remainder of the program. In object-oriented languages, methods can only directly access or make changes to an object with which they are associated. Methods are considered as a provider of service to an object and make changes to the object in a way consistent with the object's intended behavior.

2.2 Sequential Program- Multiplex of methods

The execution of a method is dependent upon another method's execution if a variable (memory address(es)) in its read set is directly (or indirectly in a transitive closure) in the write set of the other method. Dependences between methods result in a partial ordering of the methods, and this partial ordering determines their execution order. A sequential program is a multiplexing of the methods of a program into a total order. The total ordering of the methods provides a convenient but implicit way to specify the dependence relationships: a method that is dependent upon another method is placed (and thereby, executed) after the predecessor method in the sequential program.

Methods in procedural and object-oriented programming languages can have side-effects, i.e., a method can make changes to the global program state (its write set) that might not be easily analyzable statically. These potential side effects create unknown dependencies. If static analysis of a method's side effects (or, lack thereof) is not possible, the compiler assumes that all methods might have side effects and that a method could be dependent upon any prior method. This implies that the methods should be executed in the total order in which they are arranged by the compiler in the sequential program.

However, many practical considerations limit the side effects of a method. First, it is not always likely that a given method will modify program state accessed by another arbitrarily-chosen method. This is especially true with applications that have unstructured parallelism often exhibited due to the nature of application, good software engineering practices and usage of object-oriented programming concepts. Second, modern languages (especially, object-oriented languages) are evolving with strict programming specifications and require modular programs with methods associated with a class object. Changes made by a method are visible only to the other methods associated with them. Therefore, two methods with different associations and/or with disjoint read sets are independent, and could possibly execute in parallel, even though they are ordered in the sequential program. We next present some motivating examples to illustrate PD.

2.3 Motivating Examples

The methods chosen for PD could be from system libraries, application libraries, modules used by a program, the class objects in a program, or from the program itself. Methods from system libraries, such as calls to the memory allocator, file buffer, and network packet operations, are usually low-level methods that sometimes trap to the operating system to finish their task. The execution of these methods rarely interferes with the program except for the parameters and value returned by the method. For example, `malloc` (or `new`) is one of the most frequently executed methods. The program calls `malloc` with the amount of memory needed. The method can be demultiplexed and concurrently executed with the program; its ordering with other memory allocator calls (such as `free`, `resize` and other methods that modify book-keeping structures of memory allocation) is the only requirement for correct execution. The parameter passed to `malloc` can be determined by executing the computation in the program that creates the parameter (which could create a dependency with the program hence limiting the concurrency) or by predicting it. The latter is practical with memory allocation because programs tend to require memory of a small set of sizes defined by the data structures they are using. For example, in the SPEC CPU2000 benchmarks, 95% of the calls to `malloc` are with parameters that have been used to call `malloc` before. Multiple executions of `malloc` can also be triggered with different sizes as parameters, ultimately using the appropriate result when called.

Methods for PD in application libraries and programs could be those associated with implementations of abstract data structures such as linked lists, binary trees, heaps, B-Trees, and hashes that access and modify the data according to its semantics, methods that implement service functions for the application, and so on. The following are some examples from the SPEC CPU2000 benchmark suite.

`254.gap` implements a language and library for computation used in group theory. In the following example, we consider the method `NewBag` for demultiplexing. The method is invoked from as many as 500 different locations in the program, and contributes 17% of the total run time – 7% from `NewBag`, and 10% from the `CollectGarb` method that is executed within `NewBag`. `NewBag` takes two parameters: the type of bag to be created and its size. The `'type'` parameter can take 30 possible types but is limited to very few depending on the method that calls `NewBag`. The `'size'` parameter can also be easily identified depending on the `'type'`. For example, the `'size'` is always four when `'type' = T_LIST`. In our run with train inputs, the method was invoked 6.8 million times and for 99% of the calls the parameters used were the same as that of a previous call.

`175.vpr` is a FPGA placement and routing application. It spends 86% of its run time in operations on its heap data structures. 8% of its run time is from `alloc_heap_data` (Figure 2(a) and (b)), a method to

allocate memory in the heap structure. The program spends the rest of the 86% in `get_heap_head`, `expand_neighbours`, `node_to_heap`, and `add_to_heap` methods. The application calls these methods to alter the value of elements in the heap, get the head of heap, and insert a new node onto the heap. We illustrate PD with the simple example of method `alloc_heap_data`. The method allocates a chunk of data if `heap_free_head` is not set; otherwise, it recycles the chunk of memory recently freed by the method `free_heap_data`. We can demultiplex `alloc_heap_data`, begin its execution either when its previous execution result is used or when `heap_free_head` is called.

`186.crafty` is a computer chess program. It spends its execution time evaluating the chess board, planning its moves, and eventually making them. We find a number of methods that can benefit from PD such as `AttacksTo`, a method where the application spends 6% of its time (Figure 2(c) and (d)). The `AttacksTo` method is used to produce a map of all squares that directly attack the specified square and is called by several methods in the program; `ValidMove`, which is used to verify that a move is valid, is one of them. As it can be seen from the figure, the `AttacksTo` method is being called repeatedly with easily identifiable parameters. The execution of `AttacksTo` can be made to begin at the beginning of execution of the `ValidMove` method. The execution could begin even earlier (i.e. immediately after the read set – the state of the chess board is available) with additional checks that determine if `AttacksTo` will be called.

2.4 Program Demultiplexing framework

The goal of Program Demultiplexing is to create concurrency in a program by executing many of the methods early, and in parallel with the rest of the program (or with other methods). To do this first requires the (re-) creation of a partial ordering of a program's methods from the sequential total ordering, i.e., a demultiplexing of methods from their multiplexed order. Once a likely partial execution ordering has been established, the program can launch methods for possible execution in parallel. The methods can then speculatively execute and buffer the results of execution for later use by the program (or by other methods). When the method's call site is reached in the sequential program, the results of the prior speculative execution can be used if they are still valid.

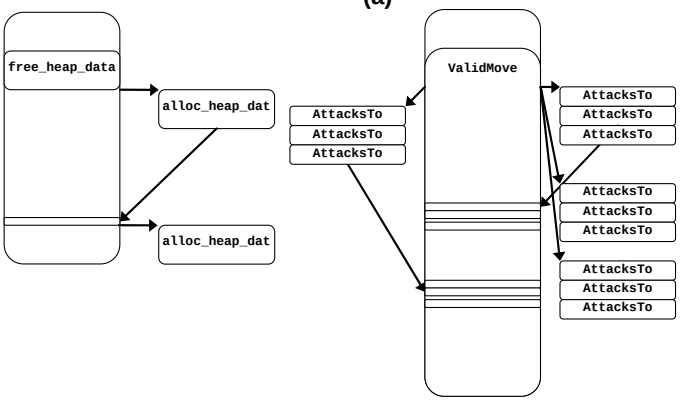
Figure 3 illustrates the concept of Program Demultiplexing and the steps required for its possible implementation. Suppose method M of the program has been chosen for demultiplexed execution. First, the parameters, if any, are generated and the method launched for execution in a different processing core. M then executes, recording the parameters, the read set it uses for execution, and buffers (i.e., does not commit) the return value and the set of changes that will be visible to the

```
static struct s_heap *alloc_heap_data (void) {
 if (heap_free_head == NULL) {
    /* No elements on the free list */
    heap_free_head = my_malloc (NCHUNK * sizeof (struct s_heap));
    . . .
 }
 temp_ptr = heap_free_head;
 heap_free_head = heap_free_head->u.next;
 return (temp_ptr);
```

```
static void node_to_heap (…
 hptr = alloc_heap_data ();
 . . .
 add_to_heap (hptr);
}
```

```
static void free_heap_data (hptr){
 hptr->u.next = heap_free_head;
 heap_free_head = hptr;
}
```

(a)



(b) **(d)**

```
int ValidMove (ply, wtm, move) {
. . .
. . .
    case king:
      if (abs(From(move)-To(move)) == 2) {
         . . . if ((!(WhiteCastle(ply)&2)) ||
                   And(Occupied,Shiftr(mask_3,1)) ||
                   And(AttacksTo(2),BlackPieces) ||
                   And(AttacksTo(3),BlackPieces) ||
                   And(AttacksTo(4),BlackPieces)) . . .
           else if . . .
                   And(Occupied,Shiftr(mask_2,5)) ||
                   And(AttacksTo(4),BlackPieces) ||
                   And(AttacksTo(5),BlackPieces) ||
                   And(AttacksTo(6),BlackPieces)) . . .
      . . .
                   And(Occupied,Shiftr(mask_3,57)) ||
                   And(AttacksTo(58),WhitePieces) ||
                   And(AttacksTo(59),WhitePieces) ||
                   And(AttacksTo(60),WhitePieces)) . . .
      . . .
                   And(Occupied,Shiftr(mask_2,61)) ||
                   And(AttacksTo(60),WhitePieces) ||
                   And(AttacksTo(61),WhitePieces) ||
                   And(AttacksTo(62),WhitePieces)). . .
```

```
BITBOARD AttacksTo(square) {
register BITBOARD attacks;
  . . .
attacks=And(w_pawn_attacks[square],BlackPawns);
attacks=Or(attacks,And(b_pawn_attacks[square],WhitePawns));
attacks=Or(attacks,And(knight_attacks[square],Or(BlackKnights,
                                                 WhiteKnights)));
attacks=Or(attacks,And(AttacksBishop(square),BishopsQueens));
attacks=Or(attacks,And(AttacksRook(square),RooksQueens));
attacks=Or(attacks,And(king_attacks[square],Or(BlackKing,
                                                 WhiteKing)));
. . .  . . .
return(attacks);
}
```

(c)

Figure 2. Example of PD. 175.vpr (`alloc_heap_data`) (a) source code and (b) PD illustration. 186.crafty (`AttacksTo`): (c) source code and (d) PD illustration.

global state (its write set). Simultaneously, the program executes and updates to the read sets of M are monitored. If an update is detected before M is called by the program, the read set has been violated and hence, the results obtained from the execution are discarded. When the call site for M is reached in the program, the results of the execution are used if they have not already been invalidated, *and* if the parameters that were used for the execution match the ones in the call site. The steps needed to implement the concept of Program Demultiplexing are listed below, with implementation details discussed in Section 3.

1. A *handler* is associated with every call site of a method chosen for demultiplexing. The handler allows separating the execution of the method from the call site in a program which is written in an imperative language and compiled for sequential execution. It sets up the execution, may execute the method one or more time(s), depending on the control flow in the handler, and provides parameters for the execution(s).

2. A *trigger* is also associated with every call site of a method chosen for demultiplexing. The site of the trigger associated with a method is different from its call site. The trigger begins the execution of the associated handler. It is constructed to usually fire after the *handler and method* are ready, i.e., their read sets are available.

3. The speculative execution of the handler and method is scheduled on an auxiliary processor. The read set accessed during each of the method's executions along with the handler is separately recorded. A hardware implementation may also choose not to invoke the demultiplexed execution or abort if sufficient resources are not available or in case of system events such as interrupts.

4. The program state that the method modifies, also known as the write set, and its return value (usually stored in a register), forms the results of the execution. The results of every execution are tagged with method's call site and its parameters (if any) and *buffered* (not committed) in an *execution buffer pool*. The handler's changes are used by the execution, but are not part of the execution's write set.

5. The execution of a method is *invalidated* when: (a) the read set is violated by a write in the program, (b) the parameters used in the execution and in the call do not match, or (c) other serializing events such as interrupts occur in the hardware.

6. The execution results are communicated from the execution buffer pool to the program (which runs on the main processor) or to another demultiplexed method (nested method calls) when a call site is reached. An execution will not be used if the program (or another method) does not call that method. When the call site is reached and the execution is still ongoing, the program may stall, waiting for the demultiplexed execution to complete, or instead abort the execution and execute the method on the main processor.

3. Implementation

We discuss the implementation details of handlers in Section 3.1, triggers in Section 3.2, and the hardware support required in Section 3.3.

3.1 Handlers

A demultiplexed execution of a method is initiated by first executing the handler. The handler has to perform relevant tasks that the call site in the program performs before calling a method, so that the method can be separated from the call site. This task is usually to set up the execution by providing parameters, if any, for the method. Therefore, we construct handlers by *slicing* (subsuming) some part of the code before a call site.

3.1.1 Slicing

The stack is an important structure for sequential programs (especially in C and C++ programs which we focus on in this work). The program code before the call site often consists of generating the parameters and putting them on the stack. The method, when called, accesses the parameters from the stack and performs its computation. Clearly, to achieve separation of a demultiplexed execution from its call site, we need to include the call site's program code, that generates parameters, in the handler. However, in many programs (as observed with the SPEC integer benchmarks) this fragment of code cannot be easily demarcated. To simplify this issue, we assume that the computation before the call site to generate parameters would use the stack exclusively. We therefore compose handlers by slicing *backward dependence chains* of the parameters from the

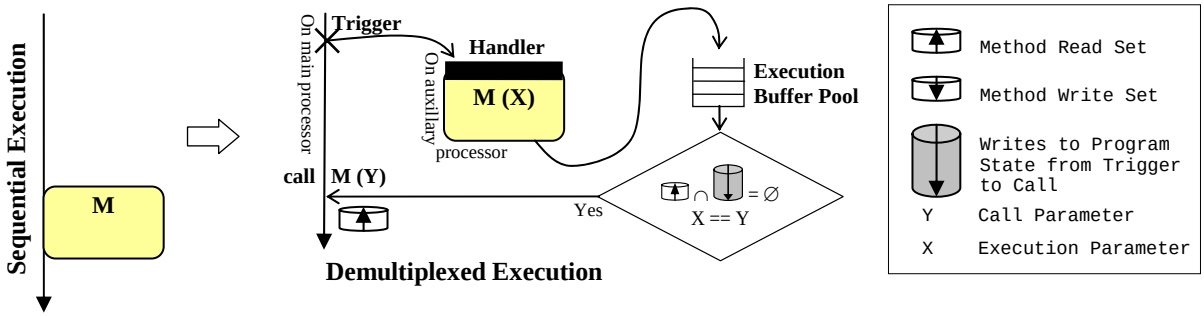


Figure 3. Program Demultiplexing framework

program, terminating them when they reach loads to the global program state (i.e., heap). Any stores to the program state that may happen during this process are *not* included in the handler (see Figure 4(c)). As a result, the handler may not be able to include all near-by computation of parameters which may prevent the overlap of demultiplexed execution with the program. On the other hand, the handler may include a large number of instructions due to the pervasive use of the stack in the program, introducing significant overheads in the demultiplexed execution. Trade-offs such as this are considered when generating handlers. In addition, control-flow and other method calls (interprocedural dependencies) encountered will likely have to be (unless, optimized) included in the handler and a brief discussion follows. The experimental details of generating a handler are discussed in Section 4.1.

Control-flow. In the example shown in Figure 4(a), method M is chosen for demultiplexing, and one of the call sites of M is shown to be present in method C. Method C's control-flow is shown to have an if-then-else structure, wrapped in a loop. Basic blocks in C are numbered 1, 2, 3, and 4. M is called from block 2, which is where the backward slice generation for the handler starts. Simple assignment of constants or arithmetic evaluation for the parameters found here can be subsumed in the handler. The control-dependency of the call site with other blocks introduces branches and loops in the handler. In the example, the branch in block 1 and the loop structure are included. In the case when M is called on both paths of a hammock (Figure 5(b)), two separate handlers are generated for the two call sites. Each handler evaluates the branch in block 5, and may (or may not) call M depending on the outcome.

Interprocedural. A value in the slice may be live earlier than the method in which the call site is present (in the example shown in Figure 5(a), method C). In this case, the slicing can continue to be performed on the call site of that method. However, this is not possible (in the unoptimized case), with a single handler when there are multiple call sites, as in the example where C can be called by both A and B.

3.1.2 Example and Optimizations

Figure 5(a) gives an example of code from 300.twolf. The generated handler for method add_penal (call site at last line) is listed in Figure 5(b). Statements included in the handler are highlighted in the source code listing.

Numerous opportunities exist for choosing and optimizing handlers (none of which we have implemented currently). First of all, programmers, with support from the programming language, may be able to express how the method can begin its demultiplexed execution. With additional book-keeping state of a method's executions, a handler can predict its parameters. The code for predictions may be based on the parameters passed during previous calls to the method, the call site, the program path that led to the call, and so on. Other optimizations include: (a) having multiple call sites in one handler, (b) optimizing a handler by omitting infrequently executed program paths to eliminate some program dependence and increase the opportunity for hiding demultiplexed execution with the rest of program, and (c) using handlers for other optimizations such as slicing of other global dependences in a method.

3.2 Triggers

A trigger is associated with every call site of a demultiplexed method. Like handlers, triggers are generated by software, with additional hardware support for evaluating them. A trigger is conceptually composed of one or more predicates joined by logical operators (such as AND and OR). When the expression of the trigger evaluates to true, it is said to have fired and begins demultiplexed execution of the handler on a processing core. The predicates supported in the current implementation are: (i) a *program counter predicate* of the form `(PC == X)` which evaluates to true when the program counter of committed instruction is `X` and (ii) a *memory address write predicate* of the form `(STORE_ADDR == Y)` which evaluates to true when the program writes to memory address `Y`. The steps required for constructing the trigger are described next.

First, we *annotate* all memory write instructions in the program, the memory read operations in a given demultiplexed method and in the instructions included in

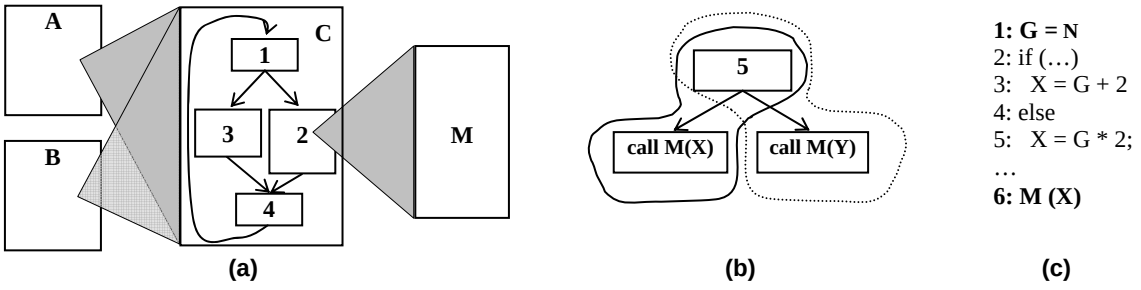


Figure 4. Generating Handlers for Method M. (a) Basic block 2 in C calls M. C can be called by A and B. (b) M is called in both the paths of a hammock. Two separate handlers are constructed in the unoptimized case. (c) Example code segment where variable G is global and therefore, line 1 is not included in the handler which hinders the overlap of demultiplexed execution

the handler for a particular call site of that method. Annotations log the program counter, memory addresses, and whether they are stack or global data addresses. The program with the annotations is executed to create a *profile***.**

For a given call site, the read set R of the method and its handler for the execution are collected from the profile. This includes unmatched global heap references and any unmatched stack references (which will exist depending on the backward slice terminating conditions for generating the handler). We then identify when all the references in the read set R are available in the program from the program's profile. This point in the program, called the *trigger point*, is the earliest that the method can execute without violating data dependencies, and is collected for several executions of the method. Figure 5(c) illustrates this step.

The trigger points are then studied and the set of predicates for the trigger are chosen so that demultiplexed execution, when begun, will rarely, if ever, be invalid. The predicates may be chosen to allow earlier execution which may result in more overlap, but may also increase the number of invalid executions.

3.3 Hardware Support

We now discuss the hardware support required for evaluating triggers, performing demultiplexed executions, storing and invalidating executions, and using the results of valid executions. A thorough discussion is not provided in this paper due to space constraints and is left for future work.

Triggers. The generated triggers are registered with the hardware for evaluation. For this, we require extensions to the instruction set architecture and storage of predicates. The predicates are evaluated based on the (logical address of) program counters and memory write addresses of instructions committed by the program. The search can be effectively implemented with minimal overheads by means of filtering (for example, Bloom filters [5]).

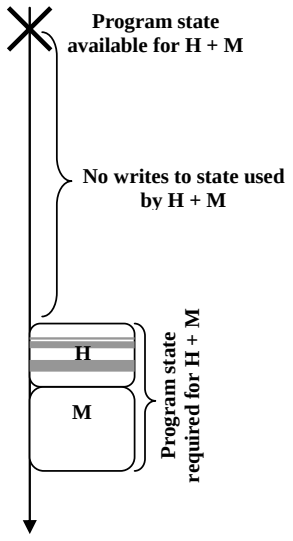
Demultiplexed execution. When triggered, demultiplexed methods are scheduled on available auxiliary processors on a first-come first-serve basis. A part of the cache hierarchy is used to store the results during the demultiplexed execution. (We assume a typical multiprocessor system.) Cache lines that are used for execution are augmented with *access* bits, used to identify any cache line references made during the execution. When a trigger is fired, the program counter of the associated handler is communicated to an available auxiliary processor. The access bits are cleared, the processor switches to speculative mode during which writes do not send invalidate messages to other processors and do not request exclusive access of cache lines. The changes in a handler should be stored and provided to the speculative execution for a demultiplexed method, but discarded at the end of execution. The access bit is set when the corresponding cache line is accessed. Eviction of a dirty line from the cache terminates execution as it indicates lack of hardware resources to buffer execution. At the end of the execution, the following operations are performed: (i) the read set, which is the set of all

```
delta_vert_cost = 0
acellptr = carray[ a ]
axcenter = acellptr->cxcenter
aycenter = acellptr->cycenter
aorient  = acellptr->corient
atileptr = acellptr->tileptr
aleft    = atileptr->left
aright   = atileptr->right
atermptr = atileptr->termsptr
bcellptr = carray[ b ]
bxcenter = bcellptr->cxcenter
bycenter = bcellptr->cycenter
borient  = bcellptr->corient
btileptr = bcellptr->tileptr
bleft    = btileptr->left
bright   = btileptr->right
btermptr = btileptr->termsptr
newbinpenal = binpenal
newrowpenal = rowpenal
newpenal    = penalty
new_old( bright-bleft-aright+aleft )
find_new_pos() ;
a1LoBin = SetBin(startxa1 = axcenter + aleft)
a1HiBin = SetBin(endxa1   = axcenter + aright)
b1LoBin = SetBin(startxb1 = bxcenter + bleft)
b1HiBin = SetBin(endxb1   = bxcenter + bright)
a2LoBin = SetBin(startxa2 = anxcenter + aleft)
a2HiBin = SetBin(endxa2   = anxcenter + aright)
b2LoBin = SetBin(startxb2 = bnxcenter + bleft)
b2HiBin = SetBin(endxb2   = bnxcenter + bright)
old_assgnto_new2(a1LoBin, a1HiBin, b1LoBin,
sub_penal( startxa1, endxa1, ablock, a1LoBin
sub_penal( startxb1, endxb1, bblock, b1LoBin
add_penal( startxa2, endxa2, bblock, a2LoBin
add_penal( startxb2, endxb2, ablock, b2LoBin,
b2HiBin)
```

(a)

```
;; a, b, ablock, block
;; are global variables
mov    0x80e9468,%ebx
mov    0x80e9414,%ecx
mov    0x80e9730,%edi
mov    0x80e9780,%edx
mov    %edx,0xffffff7c(%ebp)
mov    0x80e9778,%esi
lea    (%edi,%esi,1),%edi
mov    %edi,%eax
sub    %ebx,%eax
cltd
idiv   %ecx
mov    %eax,0xffffffa8(%ebp)
mov    0xffffff7c(%ebp),%eax
add    %eax,%esi
mov    %esi,%eax
sub    %ebx,%eax
cltd
idiv   %ecx
mov    %eax,%ebx
mov    %esi,0x4(%esp,1)
mov    0xffffffa8(%ebp),%ecx
mov    0x80e97cc,%eax
mov    %edi,(%esp,1)
mov    %ecx,0xc(%esp,1)
mov    %ebx,0x10(%esp,1)
mov    %eax,0x8(%esp,1)
call   8049ab0 <add_penal>
```

(b)



(c)

Figure 5. Example of Handler and Generating Triggers. (a) Program code from 300.twolf. Lines in the handler generated for the `add_penal` method (last line) are highlighted, (b) Generated Handler. (c) Identifying the trigger point for an execution of method M.

Table 1. Experimental Evaluation

System	Virtutech Simics 2.0. Multiprocessor system with upto six Intel Pentium 4 processors on an 875P chipset with IDE disks running Debian Linux (Kernel 2.6.8).
Processor Core	3-GHz out-of-order 4-wide superscalar processor with 7 pipeline stages. No cracking of instructions to micro-ops. 64-entry reorder buffer. 1024-entry YAGS branch predictor, 64-entry return address stack. Load instruction issues only after all prior stores are resolved.
Memory System	Level-1 private instruction cache is 64-KB, 2-way, with 2-cycle hit latency with fetch buffer that prefetches the next line. Level-1 private data cache is 64-KB, 2-way with 3-cycle hit latency, write-back and write-allocate with MSI states for cache coherence. Level-2 private inclusive and unified cache is 1-MB, 4-way, with 12-cycle hit latency and MSI states for cache coherence. Line size is 64 bytes for all caches. Cache-to-cache transfers take 12-cycles. 512 MB DRAM with 400 cycle access latency.
PD Execution	First-come first-serve policy of scheduling demultiplexed executions on first-available processor. 5-cycles to communicate the program counter of the handler. Communication with buffer pool takes 15-cycles. Up to 4-lines transferred concurrently.

"accessed" but not dirty cache lines, and the write set, which is the set of all "dirty" cache lines, are identified and sent to an execution buffer pool, (ii) the dirty cache lines are marked invalid, and (iii) the processor returns from speculative mode. As it may be expensive to scan the cache for the read and write sets, additional set of buffers or filters can be used to identify accessed and dirty cache lines. Such optimizations have been used in previous speculative parallelization proposals (for example, [41]).

Storage of executions. The identified read and write sets of a demultiplexed execution are named with the method's call site and its parameters and stored in an execution context in the buffer pool. The execution buffer pool is an additional cache-line based storage required to hold the results of various demultiplexed executions until their use or invalidation. The execution buffer pool is a central (shared) hardware structure in our current implementation. Additional logic to invalidate executions on violations is described next.

Invalidating executions. All committed program writes occurring in the sequential program are sent to the execution buffer pool, in addition to the invalidates that are sent to other processors as a part of the coherence mechanism. The read set of the executions in the buffer pool are searched for the program write address to identify those that violated data dependencies. This search can be efficiently implemented with address filters and can be pipelined into many stages, if needed. An invalidate request received by a processor with an on-going demultiplexed execution on an "accessed" cache line terminates and invalidates the execution. Some system events such as timer and device interrupts can also terminate an ongoing demultiplexed execution.

Using executions. Results of a demultiplexed execution can be used by the program or in the case of nested method calls, by another demultiplexed execution. On occurrence of a call, a search operation is initiated in the execution buffer pool for the given method and on auxiliary processors that may have an execution ongoing. In the latter case, the processor may decide to stall until the execution completes and then use the results, or perform the execution of the method itself, hence aborting the demultiplexed execution. If a valid execution is found in the execution buffer pool (if there are more than one, the first one is used), the write set is copied into the processor's cache that is requesting the results. This action leads to the committing of the results when performed by the main processor, or speculative integration of the results if requested by an auxiliary processor. To reduce the overhead of committing the results of an execution, the main processor may continue executing the rest of program exploiting method-continuation based parallelism [7, 45].

4. Evaluation

4.1 Infrastructure

Full-system Simulator. The evaluation in this paper is based on a full-system, execution-based timing simulator that simulates multiple processors based on Virtutech Simics and the Intel x86 ISA. Table 1 describes the parameters in the simulation infrastructure. An important aspect of this work is implementing speculative execution of a sequential application in the presence of an operating system on a full-system simulator. Running arbitrary code (in our case, speculative executions) on OS-visible processors without the operating system's knowledge is catastrophic to the system (OS may panic and crash). The issues that we consider include: handling of intermittent timer and device interrupts, dealing with OS task switches, handling of TLB misses, exceptions, and system calls in demultiplexed executions, all of which require the operating system to be aware of the speculative execution, and handle it if possible.

Software Toolchain. We use the Diablo toolset [44] and modified binutils and gcc compiler tool chain for extracting debugging information, and reconstructing basic blocks, control-flow graphs, and program dependence graphs from the application binary. They are exported into the simulator which, along with dynamic profile information, is used to generate handlers and construct triggers automatically. Our current implementation is able to handle control-flow and interprocedural dependencies encountered in the program, as well as other obscure architectural features such as Intel/x86's stack style usage of floating-point registers. Since handlers are obtained directly from the application

Table 2. Program Demultiplexing implementation statistics

Programs	Dynamic instructions from method			Dynamic instructions from handler			Ratio of demultiplexed to local execution time			Cache lines written			Cache lines read			Buffer pool entries		Cycles held in the pool	PC triggers	
	Represented as Minimum / Average / Maximum															*Avg/Max*		*Avg*	*Avg/Max*	
Columns →	*2*			*3*			*4*			*5*			*6*			*7*		*8*	*9*	
crafty	14	300	2K	2	42	240	0.6	1.8	4.0	0	3	13	2	8	42	15	52	900	2	18
gap	11	80	245	0	8	32	0.6	2.2	3.8	1	5	11	1	8	19	13	32	590	2	12
gzip	28	62	180	0	7	20	1.2	1.5	2.0	2	4	7	7	10	14	1	8	70	1	2
mcf	11	13	543	0	12	20	3.0	5.0	7.0	2	18	83	3	24	111	4	28	520	3	7
parser	10	55	455	0	9	79	0.5	2.0	3.9	2	10	94	5	10	12	12	52	413	3	14
twolf	72	455	1K	0	16	37	1.0	1.4	2.1	2	7	17	2	22	60	3	9	244	2	4
vortex	35	45	63	9	18	38	0.5	1.3	5.0	3	6	11	6	13	22	2	16	160	2	27
vpr	28	220	945	1	9	32	1.0	1.0	1.1	2	3	18	1	7	22	4	23	308	1	8

binary, manipulations to the stack pointer (such as push, pop, call, and ret instructions) encountered while slicing are included. This ensures that the parameters are written in the position on the stack that the instructions in the method will read them from. Our implementation of triggers uses only program counter predicates. Other forms of predicates will be useful for optimizing the triggers.

Benchmarks. We use 8 of the 12 integer programs in SPEC CPU2000 benchmark suite compiled for the x86 architecture using the GNU gcc compiler version 3.3.3 with optimization flag `-O2 –fno-inline –fno-optimize-sibling-calls`[2]. (The other benchmarks in the suite are not supported by our toolset.) The benchmarks are run with train inputs for 200 million instructions after the initialization phase.

4.2 Results

We begin the discussion by presenting the potential for PD, which is the ability to begin the execution of a method well before its call site in the sequential program. Figure 6 plots the ratio of the cycles elapsed between when the method and its handler are ready to execute, i.e., the trigger point, and when the method is called by the program, to the cycles it takes to execute the method. We assume the execution of the method in an auxiliary processor with no overheads – as if the execution happened on the main processor running the program. In the case when a method invokes another method (nested calls), we include all program references made by that nested method only if it is not considered for demultiplexing. Each point in the graph indicates a call site of a method. All call sites of methods with greater than 2% of the total execution time are evaluated for this study. The Y-axis presents the ratio on a logarithmic scale. The ratio gives an idea of how well a demultiplexed execution could be overlapped. A ratio close to 0 indicates the read set of an execution is available only just before the method is called, therefore no overlap is possible. A ratio close to 1 indicates the method can be executed immediately after the read set is ready; it may finish execution just before it is called when overheads are considered. A ratio >> 1 indicates ample

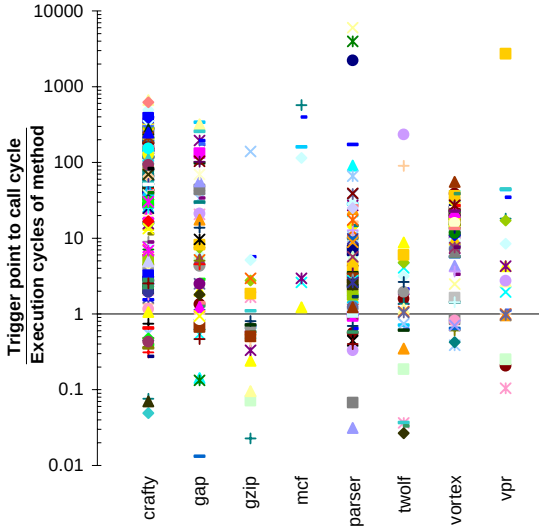


Figure 6. Potential for PD. Ratio of cycles between the trigger point and the call site, to the execution cycles of the method without overheads

[2] Intel x86 program binaries use the stack for passing parameters. GNU gcc provides a flag `–mregparm=N` that allows using N registers for this purpose instead. However, it requires recompilation of the entire system (libraries) with the same flag, as this optimization is not commonly used. This is beyond our reach. Another flag `-funit-at-a-time` introduced in gcc 3.4, among other optimizations, uses registers for passing parameters within a compilation unit of functions. This improves performance only by ~1% and therefore, we do not use it.

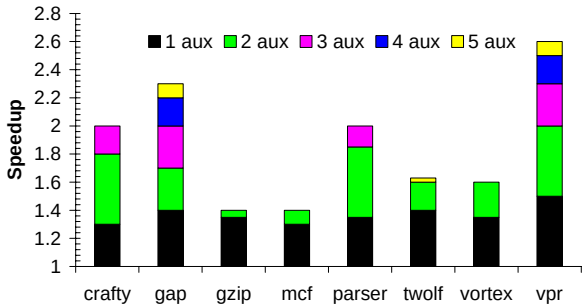


Figure 7. Performance evaluation

opportunity for completely overlapping the execution even with overheads. From the graph, we see that many call sites could begin execution well before they are called.

In Table 2 we present several statistics from our current implementation of PD. Column 2 presents the minimum, average, and maximum number of dynamic instructions executed per demultiplexed execution. Column 3 presents the minimum, average, and maximum (dynamic) instructions in the generated handler. Handlers contribute 4% to 100% of additional instructions to the demultiplexed execution of a method.

Column 4 presents the overheads of demultiplexed execution, introduced due to the setting up of execution, the execution of the handler, and the additional cycles spent in the cache misses since the cache lines accessed by the demultiplexed method are not available in the local cache. However, they benefit from valid cache lines in the local cache accessed by previous demultiplexed executions in the same processor. We further minimize the overheads by prefetching the lines used in the previous execution of that method. This however, is not useful for methods that require different program state during each of its demultiplexed executions. Benchmark `mcf`, has 5x overhead because of small (in terms of number of dynamic instructions) methods that frequently cache miss when accessing different nodes of a dynamically allocated data structure. For other benchmarks, the average overheads are between 1x to 2x. We observe that some demultiplexed executions of large methods finish faster than when executed on the main processor due to lower cache conflict misses.

Column 5 and 6 present the minimum, average, and maximum number of cache lines written and read by demultiplexed executions. They denote the write and read set respectively. Column 7 presents the average and maximum number of entries used in the execution buffer pool. Column 8 presents the average number of demultiplexed executions held in the buffer pool, sampled every 50 cycles. Demultiplexed executions that are ongoing when requested by the main processor are not held in the execution buffer pool. Finally, the last column presents the average and maximum number of predicates that are required for constructing triggers. To prevent mis-speculations in our implementation, we disable demultiplexed execution of methods if the triggers we identify are not stable and may lead to invalid execution results.

We present speedup in Figure 7 with one to five auxiliary processors for demultiplexed executions (i.e., a total of two to six processors). Speedups range from 1.3x to 2.7x. Benchmarks such as `crafty`, `gap`, `parser`, and `vpr` benefit from more than two (but less than four) auxiliary processors with speedups of greater than or equal to 2. Other benchmarks achieve speedups of 1.4x due to limited opportunities. There are several possiblities to further speed up the programs ranging from optimizations to the current implementation of PD, compiler optimizations and source code changes for more opportunities, and further parallelization with other speculative parallelization proposals.

5. Related Work

Multiscalar is an early proposal for the speculative parallelization of sequential programs [38]. The hardware speculatively executes tasks in parallel that are created by statically partitioning the control flow graph. Other recent proposals [1, 8, 9, 17, 18, 25, 28, 30, 39, 40, 43] limit parallelization to specific portions of control flow graph such as loops and method-continuations and/or use data speculation to facilitate this kind of parallelization. Marcuello and Gonzalez [26] and Renau et al. [33] observed that out-of-order instantiation of speculative threads in the scope of such proposals improves performance significantly. TCC [18] proposed using transactions [19], specified by programmers for control-flow speculative parallelization of sequential programs and also allowed unordered commits of transactions. Unlike these speculative parallelization models that sequence speculative threads according to the program's control-flow, PD achieves speculative data-flow style concurrent execution of a sequential program by means of triggers and handlers derived from the program. PD can therefore, execute distant parts of the program well ahead of the control flow, rarely mis-speculating, if ever.

Lam and Wilson [23] evaluated the limits of control flow on speculative parallelization. Warg et. al [45] evaluated the limits of speculative parallelization of methods in imperative and object-oriented programming languages. Martel et. al [27] presented different parallelization strategies to exploit distant parallelism in the SPECint95 suite. The Mitosis compiler [32] speculatively begins execution of loops with *fork* instructions inserted in the program and generates live-ins for the threads by executing backward slices of instructions from the loop to the site of fork. Handlers and triggers in PD are conceptually similar but are unrestricted (fork in Mitosis is placed in the same function at the same basic-block level) and broader in scope, which leads to unordered speculative executions. The master-slave speculative parallelization model [50] divides a program into tasks, executing a distilled program

generated with aggressive optimizations for the common case, and verifying their execution by concurrently executing the unoptimized task achieving slipstreamed execution [42].

Helper thread models [10, 14, 24, 35, 49] are used to mitigate performance degrading events such as cache misses and branch mis-predictions in a processor by creating helper threads, which are dependence chains that lead up to the event, and executing them well before they are reached by the program. Therefore, the event is overlapped with the rest of program, mitigating its harmful effects on performance.

Data-flow machines [11, 12, 16, 29] used special languages [2, 13] that excelled in expressing parallelism but have been less popular due to the lack of features widely available in imperative languages. To handle the enormous scheduling and communication overheads of fine-grained data flow architectures, Sarkar and Hennessy [36] and Iannucci [21] proposed statically partitioning a data-flow program into subprograms and executing them in a data flow order; subprograms by themselves were executed sequentially. PD also uses partitioned sub-programs (methods) written in imperative languages, and executes them according to their data dependencies.

There has been extensive work in functional languages to extract method-level parallelism automatically as they do not have any side-effects [15, 37]. MultiLisp [34] implements parallel evaluation of parameters that allow programmers to explicitly express the concurrency of a method. Knight [22] presented speculative parallelization of Lisp programs that allows methods to have side-effects.

PD resembles message-passing based parallel programming paradigms such as Linda [6] and Actors [20]. Time-shifted modules [48] proposed a software-based approach to concurrently execute modules which have limited interaction with the program. Recently, object-oriented languages have introduced primitives for programmers to express and exploit such forms of concurrency [3, 4].

Triggers in PD are similar to watchpoints which are a familiar concept for debugging programs. Prvulovic and Torrellas [31] used watchpoints for memory locations to identify race conditions in multi-threaded programs. Zhou et al. [46, 47] proposed efficient hardware support for monitoring memory accesses in a program for software debugging and detecting bugs.

6. Summary

We introduced Program Demultiplexing, an execution paradigm that demultiplexes the total ordering of methods or functions —the subprograms in a program— in a sequential program into partially ordered methods that execute concurrently. Unlike sequential execution, methods are not executed at the call site, but well before they are called by the program. Call sites of methods are associated with triggers and handlers. A trigger is composed of a set of predicates based on program counters and memory writes. When fired, the execution of the handler generates the parameters of the method, if needed, and begins the speculative execution of the method on an auxiliary processor. The results of execution of demultiplexed methods are buffered, and later used by the main program or by other demultiplexed executions. The set of global data read by a demultiplexed method's execution is monitored and violation of this read set invalidates the execution.

PD exploits the data-flow distance (in terms of execution cycles) that is created due to the sequential control-flow in a program, speculatively executing distant parts of the program when its data is ready and later committing the results when the control flow reaches it. This work demonstrated the presence of opportunities even on programs in the SPEC CPU2000 integer benchmark suite, that were written with no intention of creating concurrency. We believe that wider usage of object-oriented programming languages and their strict programming requirements, will significantly increase the opportunities for PD, and make the data-driven concurrent speculative execution at the granularity of methods an apt choice for future multicore systems.

We presented motivating examples for PD, details of our current implementation of PD, as well as the results of an evaluation using program from the SPEC CPU2000 integer suite. We achieve 1.8x speedup (harmonic mean) on a system, with modest hardware support needed for triggers, buffering demultiplexed executions and detecting violations of executions; many of these features are also needed for supporting control-driven speculative parallelization.

Acknowledgements

This work was supported in part by National Science Foundation grants CCR-0311572 and EIA-0071924.

References

[1] H. Akkary and M. Driscoll, "A Dynamic Multithreading Processor," In *Proc. of the 31st Annual Intl. Symp. on Microarch.*, pp. 226-36, 1998.

[2] K. Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. Comput.*, vol. 39, pp. 300-18, 1990.

[3] N. Benton, L. Cardelli, et al., "Modern Concurrency Abstractions for C#," In *Proc. of the 16th European Conf. on Object-Oriented Programming*, pp. 415-40, 2002.

[4] G. Bierman, E. Meijer, et al., "The Essence of Data Access in Cw," In *Proc. Of the 19th European Conf. On Object-Oriented Programming*, pp. 287-311, 2005.

[5] B. H. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM*, vol. 13, pp. 422-6, 1970.

[6] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, vol. 32, pp. 444-58, 1989.

[7] M. K. Chen and K. Olukotun, "Exploiting Method-Level Parallelism in Single-Threaded Java Programs," In *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques*, pp. 176, 1998.

[8] M. K. Chen and K. Olukotun, "The Jrpm System for Dynamically Parallelizing Java Programs," In *Proc. Of the 30th Annual Intl. Symp. On Comp. Arch.*, pp. 434-46, 2003.

[9] M. Cintra, J. Martinez, et al., "Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors," In *Proc. of the 27th Annual Intl. Symp. on Comp. Arch.*, pp. 13-24, 2000.
[10] J. D. Collins, D. M. Tullsen, et al., "Dynamic Speculative Precomputation," In *Proc. Of the 34th Annual Intl. Symp. On Microarch.*, pp. 306-17, 2001.
[11] A. L. Davis, "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine," In *Proc. Of the 5th Annual Symp. On Comp. Arch.*, pp. 210-5, 1978.
[12] J. B. Dennis, "A Preliminary Architecture for a Basic Data-Flow Processor," In *Proc. Of the 2nd Intl. Symp. On Comp. Arch.*, pp. 125-31, 1975.
[13] J. B. Dennis, "First Version of Data Flow Procedure Language," MIT Laboratory for Computer Science MAC TM61, 1991.
[14] A. Farcy, O. Temam, et al., "Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes," In *Proc. Of the 31st Annual Intl. Symp. On Microarch.*, pp. 59-68, 1998.
[15] B. Goldberg and P. Hudak, "Implementing Functional Programs on a Hypercube Multiprocessor," In *Proc. Of the Third Conf. On Hypercube Concurrent Computers and Applications: Architecture, Software, Computer Systems, and General Issues*, pp. 489-504, 1988.
[16] V. G. Grafe, G. S. Davidson, et al., "The Epsilon Dataflow Processor," In *Proc. Of the 16th Annual Intl. Symp. On Comp. Arch.*, pp. 36-45, 1989.
[17] L. Hammond, M. Willey, et al., "Data Speculation Support for a Chip Multiprocessor," In *Proc. Of the 8th Intl. Conf. On Architectural Support for Programming Languages and Operating Systems*, pp. 58-69, 1998.
[18] L. Hammond, B. D. Carlstrom, et al., "Programming with Transactional Coherence and Consistency (TCC)," In *Proc. Of the 11th Intl. Conf. On Architectural Support for Programming Languages and Operating Systems*: ACM Press, pp. 1-13, 2004.
[19] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," In *Proc. Of the 20th Annual Intl. Symp. On Comp. Arch.*, pp. 289-300, 1993.
[20] C. Hewitt. Viewing Control Structures as Patterns on Passing Messages. *Journal of Artificial Intelligence*, vol. 8, pp. 323-64, 1977.
[21] R. A. Iannucci, "Toward a Dataflow/Von Neumann Hybrid Architecture," In *Proc. of the 15th Annual Intl. Symp. on Comp. Arch.*, pp. 131-40, 1988.
[22] T. Knight, "An Architecture for Mostly Functional Languages," In *Proc. of the Annual Conf. on LISP and Functional Programming*, pp. 105-12, 1986.
[23] M. S. Lam and R. P. Wilson, "Limits of Control Flow on Parallelism," In *Proc. Of the 19th Annual Intl. Symp. On Comp. Arch.*, pp. 46-57, 1992.
[24] C.-K. Luk, "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors," In *Proc. Of the 28th Annual Intl. Symp. On Comp. Arch.*, pp. 40-51, 2001.
[25] P. Marcuello, A. Gonzalez, et al., "Speculative Multithreaded Processors," In *Proc. of the 12th Intl. Conf. on Supercomputing*, pp. 77-84, 1998.
[26] P. Marcuello and A. Gonzalez, "A Quantitative Assessment of Thread-Level Speculation Techniques," In *Proceedings of the 14th Intl. Symp. On Parallel and Distributed Processing*, pp. 595, 2000.
[27] I. Martel, D. Ortega, et al., "Increasing Effective IPC by Exploiting Distant Parallelism," In *Proc. of the 13th Intl. Conf. on Supercomputing*, pp. 348-55, 1999.
[28] J. T. Oplinger, D. L. Heine, et al., "In Search of Speculative Thread-Level Parallelism," In *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques*, pp. 303, 1999.
[29] G. M. Papadopoulos and D. E. Culler, "Monsoon: An Explicit Token-Store Architecture," In *Proc. of the 17th Annual Intl. Symp. on Comp. Arch.*, pp. 82-91, 1990.
[30] M. K. Prabhu and K. Olukotun, "Exposing Speculative Thread Parallelism in Spec2000," In *Proc. of 10th Annual Intl. Symp. on Principles and Practice of Programming Languages*, pp. 142-52, 2005.
[31] M. Prvulovic and J. Torellas, "Reenact: Using Thread-Level Speculation Mechanisms to Detect Data Races in Multithreaded Code," In *Proc. of the 30th Annual Intl. Symp. on Comp. Arch.*, pp. 110-21, 2003.
[32] C. Quinones, C. Madriles, et al., "Mitosis Compiler: An Infrastructure for Speculative Threading Based on Pre-Computation Slices," In *Proc. of the 31st Annual Intl. Conf. on Programming Language Design and Implementation*, pp. 269-75, 2005.
[33] J. Renau, J. Tuck, et al., "Tasking with out-of-Order Spawn in Tls Chip Multiprocessors: Microarchitecture and Compilation," In *Proceedings of the 19th annual Intl. Conf. on Supercomputing*, pp. 179-88, 2005.
[34] J. Robert H. Halstead. MULTILISP: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 501-38, 1985.
[35] A. Roth and G. S. Sohi, "Speculative Data-Driven Multithreading," In *Proc. Of the 7th Intl. Symp. On High-Perf. Comp. Arch.*, pp. 37, 2001.
[36] V. Sarkar and J. Hennessy, "Partitioning Parallel Programs for Macro-Dataflow," In *Proc. of the 1986 ACM Conf. on LISP and Functional Programming*, pp. 202-11, 1986.
[37] K. E. Schauser, D. E. Culler, et al., "Compiler-Controlled Multithreading for Lenient Parallel Languages," In *Proc. Of the 5th Acm Conf. On Functional Programming Languages and Comp. Arch.*, pp. 50-72, 1991.
[38] G. S. Sohi, S. E. Breach, et al., "Multiscalar Processors," In *Proc. Of the 22nd Annual Intl. Symp. On Comp. Arch.*, pp. 414-25, 1995.
[39] G. S. Sohi and A. Roth. Speculative Multithreaded Processors. *Computer*, vol. 34, pp. 66-73, 2001.
[40] J. Steffan and T. Mowry, "The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization," In *Proc. of the 4th Intl. Symp. on High-Perf. Comp. Arch.*, pp. 1-12, 1998.
[41] J. G. Steffan, C. B. Colohan, et al., "A Scalable Approach to Thread-Level Speculation," In *Proceedings of the 27th Annual Intl. Symp. On Comp. Arch.*, pp. 1-12, 2000.
[42] K. Sundaramoorthy, Z. Purser, et al., "Slipstream Processors: Improving Both Performance and Fault Tolerance," In *Proc. of the ninth Intl. Conf. on Architectural support for programming languages and operating systems*, pp. 257-68, 2000.
[43] J.-Y. Tsai, J. Huang, et al. The Superthreaded Processor Architecture. *IEEE Transactions on Computers*, vol. 48, pp. 881-902, 1999.
[44] L. Van Put, B. De Sutter, et al., "LANCET: A Nifty Code Editing Tool," In *Proc. of the Sixth Workshop on Program Analysis for Software Tools and Engineering*, pp. 75-81, 2005.
[45] F. Warg and P. Stenstrom, "Limits on Speculative Module-Level Parallelism in Imperative and Object-Oriented Programs on Cmp Platforms," In *Proc. of the 12th Annual Intl. Conf. on Parallel Architectures and Compilation Techniques*, pp. 221-30, 2001.
[46] P. Zhou, W. Liu, et al., "Accmon: Automatically Detecting Memory-Related Bugs Via Program Counter-Based Invariants," In *Proc. of the 37th Annual Intl. Symp. on Microarch.*, pp. 260-80, 2004.
[47] P. Zhou, F. Qin, et al., "Iwatcher: Efficient Architectural Support for Software Debugging," In *Proc. of the 31st Annual Intl. Symp. on Comp. Arch.*, pp. 224-35, 2004.
[48] C. B. Zilles and G. S. Sohi, "Time-Shifted Modules: Exploiting Code Modularity for Fine Grain Parallelism," University of Wisconsin-Madison, Computer Sciences Dept. TR1430, 2001.
[49] C. B. Zilles and G. S. Sohi, "Execution-Based Prediction Using Speculative Slices," In *Proc. Of the 28th Annual Intl. Symp. On Comp. Arch.*, pp. 2-13, 2001.
[50] C. B. Zilles and G. S. Sohi, "Master/Slave Speculative Parallelization," In *Proc. of the 35th Annual Intl. Symp. on Microarch.*, pp. 85-96, 2002.