Interprocedural Dataflow Analysis in an Executable Optimizer

David W. Goodwin
goodwin@vssad.hlo.dec.com
Digital Equipment Corporation

Abstract

Interprocedural dataflow information enables link-time and post-link-time optimizers to perform analyses and code transformations that are not possible in a traditional compiler. This paper describes the interprocedural dataflow analysis techniques used by Spike, a post-link-time optimizer for Alpha/NT executables. Spike uses dataflow analysis to summarize the register definitions, uses, and kills that occur external to each routine, allowing Spike to perform a variety of optimizations that require interprocedural dataflow information. Because Spike is designed to optimize large PC applications, the time required to perform interprocedural dataflow analysis could potentially be unacceptably long, limiting Spike's effectiveness and applicability. To decrease dataflow analysis time, Spike uses a compact representation of a program's intraprocedural and interprocedural control flow that efficiently summarizes the register definitions and uses that occur in the program. Experimental results are presented for the SPEC95 integer benchmarks and eight large PC applications. The results show that the compact representation allows Spike to compute interprocedural dataflow information in less than 2 seconds for each of the SPEC95 integer benchmarks. Even for the largest PC application containing over 1.7 million instructions in 340 thousand basic blocks, interprocedural dataflow analysis requires just 12 seconds.

1 Introduction

Link-time and post-link-time optimizers can perform analyses and code transformations over an entire program, enabling optimizations that are not practical in a traditional compiler. Spike is a post-link-time optimizer for Alpha/NT executables, implementing Digital's executable optimization technology [Srivastava94, Wilson96]. Spike performs a variety of profile-driven optimizations including code restructuring to improve instruction cache performance [Pettis90], Hot-Cold optimization [Cohn96], and near-optimal global register allocation [Goodwin96]. Because the entire program is available, Spike can summarize the register definitions, uses, and kills that occur external to the code segment being optimized, and then use those summaries to perform optimizations that are impossible or impractical in a traditional compiler.

Figure 1 shows examples of two types of optimizations that are possible when a summary of the defined, used, and killed registers is available. In each example, summary information is shown inside of square brackets. Figure 1(a) and Figure 1(b) show code segments where Spike uses the summary information to eliminate dead code. In Figure 1(a), register Ra is used to return a value from a called procedure. However, the summary information shows that Ra is not used on return to any caller and thus the instruction defining Ra can be deleted. In Figure 1(b), registers Ra and Rb are used to pass arguments to a called procedure but the summary information shows that the called procedure does not use the argument passed in Ra. Thus, the instruction defining Ra can be deleted. Because

```
...
def Ra
ret [ used on return = ∅ ]
```

(a)

```
...
def Ra
def Rb
call [ used by call = {Rb} ]
...
```

(b)

```
def Rt
store Rt
call [ killed by call = ∅ ]
load Rt
use Rt
```

(c)

```
save Rs
...
def Rs
call [ killed by call = ∅ ]
use Rs
...
restore Rs
```

(d)

Figure 1: Examples of optimizations enabled by register summary information.

the calling procedure and the called procedure may be in separately compiled modules, these optimizations are not available to a typical compiler.

Figure 1(c) and Figure 1(d) show examples where the register summary information enables Spike to reallocate registers to increase performance. In both figures a value is live across a call instruction. In Figure 1(c) the compiler has assigned the value to caller-saved register Rt and spilled Rt around the call. The register summary information reveals that Rt is in fact not killed by the call, and so Rt does not need to be spilled around the call instruction. In Figure 1(d) the compiler has assigned the value to callee-saved register Rs, requiring a save and restore of Rs around the function. The register summary information shows that caller-saved register Rt is not killed by the call. Thus, Spike can assign the value to Rt instead of Rs and delete the save and restore of Rs. For large PC applications, call overhead, including the overhead to save and restore callee-saved registers, is high, accounting for as much as 16% of the total execution time [Cohn96]. As a result, reassigning registers so that saves and restores are eliminated can potentially provide a significant performance improvement.

Spike can only perform these and similar optimizations because the external register definitions, uses, and kills are indicated in the summary information. Preliminary results show that these optimizations consistently provide performance improvements of 5%-10%, and in some cases provide improvements of as much as 20%.

A *routine* is a sequence of instructions generated for a high-level language procedure or function, typically having a single entry and one or more exits. A register definition that reaches a routine or a register use that causes the register to be live at a routine could be separated from the routine by arbitrarily complex control flow containing multiple levels of calls, loops, recursion, etc. Thus, generating a summary of the registers defined, used, and killed external to each routine requires interprocedural dataflow analysis. Interprocedural dataflow analysis can be performed using a program's entire control-flow graph (CFG) [Srivastava93]. A CFG for an entire program is constructed by connecting the CFG representing each routine with additional arcs representing calls and returns between the routines (e.g., see Figure 2). Because the time require to perform interprocedural dataflow analysis is typically proportional to the size of the graph being analyzed [Khedkar94], reducing the size of the graph over which the dataflow analysis is performed decreases the analysis time. Moreover, because Spike is designed to optimize large PC applications, the time required to perform interprocedural dataflow analysis could potentially be unacceptably long, limiting Spike's effectiveness and applicability. To decrease dataflow analysis time, Spike uses a compact representation of a program's intraprocedural and interprocedural control flow that efficiently summarizes the register definitions and uses that occur in the program, and that is significantly smaller than the program's CFG. Using the compact representation makes it practical for Spike to analyze programs containing millions of instructions and hundreds of thousands of basic blocks.

The remainder of the paper is organized as follows. Section 2 describes how Spike summarizes the registers defined, used, and killed external to each routine. Section 3 describes the compact representation used to represent a program and the interprocedural dataflow analyses that produce the summary information for each routine. Section 4 presents experimental results, Section 5 reviews related work, and Section 6 summarizes the paper.

2 Summarizing Register Information

For many optimizations, Spike analyzes and transforms one routine at a time. To optimize a routine, Spike requires a summary of the register definitions that occur before the routine is entered, the register uses that occur after the routine exits, and the register definitions, uses, and kills that occur during calls made by the routine. Thus, for each routine, Spike requires the following dataflow information:

- *live-at-entry*: the registers live at each entrance to the routine.
- *live-at-exit*: the registers live at each exit from the routine.
- *call-used*: as seen by the caller, the registers that may be used in a called routine before they are defined.
- *call-defined*: as seen by the caller, the registers that must be defined in a called routine.
- *call-killed*: as seen by the caller, the registers that may be overwritten in a called routine.

When analyzing a routine, Spike uses the call-used, call-defined, and call-killed sets to summarize the register uses, definitions, and kills that occur during a call. Spike replaces each call instruction to routine P with a *call-summary instruction*. The registers call-used by P are used by the call-summary instruction, the registers call-defined by P are defined by the call-summary instruction, and the registers call-killed by P are killed by the call-summary instruction. Spike uses a routine's *MAY-USE*, *MUST-DEF*, and *MAY-DEF* dataflow sets to conservatively estimate the registers call-used, call-defined, and call-killed by the routine, respectively. A routine's MAY-USE set contains the registers that may be used by the routine before being defined, the MAY-DEF set contains the registers that may be defined by the routine, and the MUST-DEF set contains

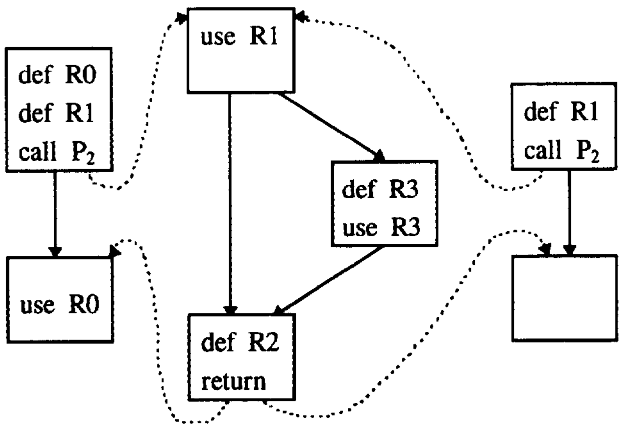

Figure 2: Example routines showing register definitions and uses.

the registers that must be defined by the routine.

Figure 2 shows the control-flow graphs for three routines, with calls and returns represented by dotted arcs. When analyzing routine P_1 or P_3 Spike uses the call-used, call-defined, and call-killed sets of P_2 to determine the register usage that occurs during the call to P_2. For P_2, call-used = MAY-USE[P_2] = {R1}, call-defined = MUST-DEF[P_2] = {R2}, and call-killed = MAY-DEF[P_2] = {R2, R3}. Thus, the call-summary instruction that replaces a call to P_2 uses R1, defines R2, and kills R2 and R3, as shown in Figure 3.

After replacing each call instruction in a routine with

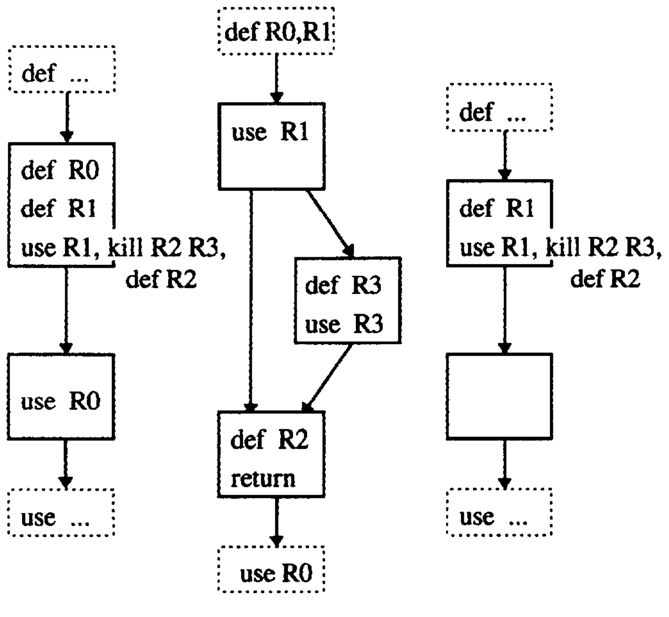

Figure 3: Routines from Figure 2 showing entry, exit and call-summary instructions.

the corresponding call-summary instruction, Spike uses a live-at-entry set to summarize the register definitions that occur before a call to the routine, and uses a live-at-exit set to summarize the register uses that occur after a return from the routine. At each entry, Spike inserts an *entry instruction* that defines the registers live at that entry, as indicated by the corresponding live-at-entry set. Similarly, at each exit, Spike inserts an *exit instruction* that uses the registers live at that exit, as indicated by the corresponding live-at-exit set. Figure 3 shows the entry and exit instructions inserted for each routine in Figure 2. The live-at-entry and live-at-exit sets include any registers that could subsequently be used before being defined along any path from the entry or exit, including all possible return paths. Thus, in routine P_2 live-at-entry = {R0, R1} and live-at-exit = {R0}. R0 is in the live-at-entry and live-at-exit sets because a return path from P_2 leads to a use of R0 in P_1.

3 Interprocedural Dataflow

To generate the interprocedural dataflow sets described above, Spike uses a *Program Summary Graph* (PSG) similar to Callahan's [Callahan88] and a two phase approach [Srivastava93] that computes the call-used, call-defined, and call-killed sets for each routine in the first phase, and then computes the live-on-entry and live-on-exit sets in the second phase. This section first describes the construction of the program summary graph, and then describes the dataflow phases.

3.1 Program Summary Graph

The program summary graph is a compact representation of a program's intraprocedural control flow, composed of *nodes* and directed *edges*. Although it is conceptually similar to the PSG proposed in [Callahan88], the PSG used by Spike differs significantly in its construction and in its encoded information. Each PSG node represents a location in the program for which dataflow information is being collected. Each node records the MAY-USE, MAY-DEF, and MUST-DEF sets for the program location represented by the PSG node. Two nodes are connected by an edge if there is a possible control-flow path between the program locations represented by those nodes. Each edge records a summary of the register definitions and uses that occur along the control-flow paths represented by the edge.

Spike examines each routine individually, producing four types of PSG nodes to represent the routine: 1) an *entry node* is produced for each entrance to the routine, 2) an *exit node* is produced for each exit from the routine, and 3) a *call node* and 4) a *return node* are produced for each call instruction in the routine. Figure 4(a) shows the control-flow graph for a routine containing four basic blocks and a single call instruction. To simplify the

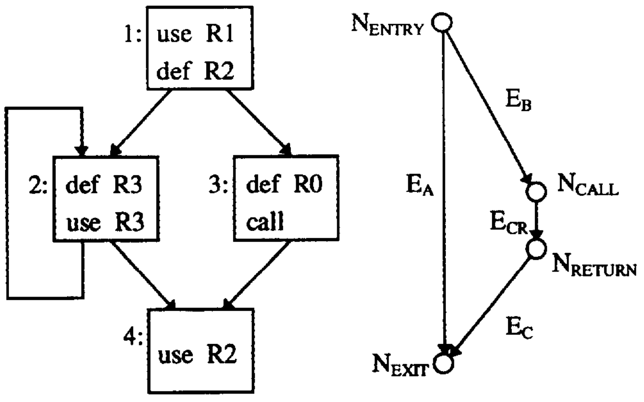

Figure 4: Example CFG and corresponding PSG nodes and edges.

presentation, the following discussion assumes a basic block is ended by a call instruction (as usual, basic blocks are also ended by branches). Figure 4(b) shows the PSG nodes produced to represent the entrance to the routine, the exit from the routine, and the call instruction in the routine. Next, Spike produces two types of PSG edges to connect the nodes: 1) a *call-return edge* connects each call node to the corresponding return node, and 2) a *flow-summary edge* connects two nodes if there is a possible control-flow path between the locations represented by the nodes. In Figure 4(b), the flow-summary edges are labeled E_A, E_B, and E_C, and the call-return edge is labeled E_{CR}.

Each flow-summary edge represents all possible control-flow paths between the locations represented by the nodes incident to the edge. Spike labels each flow-summary edge with the registers that are defined along all control-flow paths between the nodes (MUST-DEF), with the registers that are defined along some control-flow path between the nodes (MAY-DEF), and with the registers that are used before they are defined along some control-flow path between the nodes (MAY-USE). For each flow-summary edge $E = (N_X, N_Y)$, Spike constructs a subgraph of the routine's control-flow graph containing the basic blocks and arcs that are part of any path from X to Y. Each basic block is labeled with the registers defined (DEF) in the basic block, and with the registers that are used-before-defined (UBD) in the basic block. Figure 5 shows the subgraph of the CFG represented by each flow-summary edge in Figure 4(b). Node N_{CALL} represents the call instruction at the end of basic block 3. Thus, basic block 3 is represented by flow-summary edge E_B but not by flow-summary edge E_C. Basic block 4 is represented by both flow-summary edge E_A and flow-summary edge E_C because basic block 4 is on a path from the routine's entrance (represented by node N_{ENTRY}) to the routine's exit (represented by node N_{EXIT}), and on a path from the call instruction (represented by node N_{RETURN}) to the routine's exit. Similarly, basic block 1 is represented by both flow-summary edge E_A and flow-summary edge E_B.

The MUST-DEF, MAY-DEF, and MAY-USE sets for each flow-summary edge are found using conventional dataflow techniques [Aho88] on the subgraph of the CFG represented by the flow-summary edge. The dataflow equations are summarized in Figure 6. After the dataflow sets converge for flow-summary edge $E = (N_X, N_Y)$, the MUST-DEF, MAY-DEF, and MAY-USE sets associated with location X indicate the registers that must be defined, that may be defined, and that may be used along paths from X to Y. Thus, flow-summary edge E is labeled with the MUST-DEF, MAY-DEF, and MAY-USE sets associated with node N_X. For example, for the subgraph in Figure 5(a), the dataflow sets at the entry to basic block 1 are found to be: MUST-DEF$_{IN}$[1] = {R2, R3}, MAY-

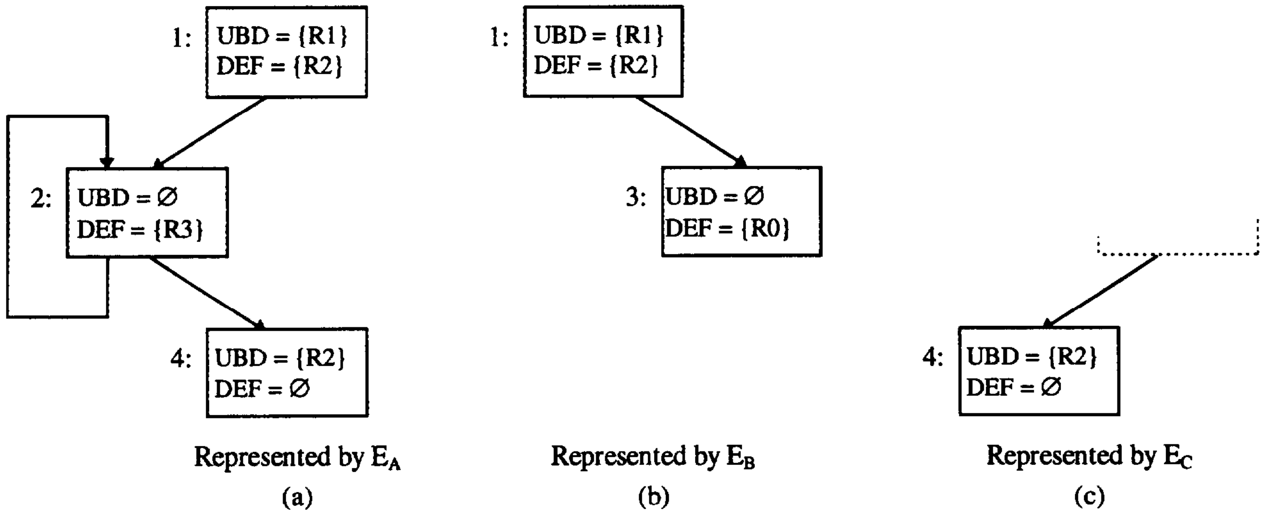

Figure 5: CFG subgraphs for the flow-summary edges in Figure 4(b).

Initialization: for each basic block B

$$\text{MAY-USE}_{IN}[B] = \text{MAY-USE}_{OUT}[B] = \varnothing$$
$$\text{MAY-DEF}_{IN}[B] = \text{MAY-DEF}_{OUT}[B] = \varnothing$$
$$\text{MUST-DEF}_{IN}[B] = \text{MUST-DEF}_{OUT}[B] = \varnothing$$

Dataflow: for each basic block B

$$\text{MAY-USE}_{IN}[B] = \text{UBD}[B] \cup (\text{MAY-USE}_{OUT}[B]-\text{DEF}[B])$$
$$\text{MAY-DEF}_{IN}[B] = \text{MAY-DEF}_{OUT}[B] \cup \text{DEF}[B]$$
$$\text{MUST-DEF}_{IN}[B] = \text{MUST-DEF}_{OUT}[B] \cup \text{DEF}[B]$$

$$\text{MAY-USE}_{OUT}[B] = \cup_S \text{MAY-USE}_{IN}[S],\ \forall \text{ succ. } S \text{ of } B$$
$$\text{MAY-DEF}_{OUT}[B] = \cup_S \text{MAY-DEF}_{IN}[S],\ \forall \text{ succ. } S \text{ of } B$$
$$\text{MUST-DEF}_{OUT}[B] = \cap_S \text{MUST-DEF}_{IN}[S],\ \forall \text{ succ. } S \text{ of } B$$

Figure 6: Dataflow equations to compute the MAY-USE, MAY-DEF, and MUST-DEF dataflow sets for flow-summary edge $E = (N_X, N_Y)$.

$\text{DEF}_{IN}[1]$ = {R2, R3}, and $\text{MAY-USE}_{IN}[1]$ = {R1}. Thus, the dataflow sets for flow-summary edge E_A are assigned as shown in Figure 7. Figure 7 also shows the dataflow sets for flow-summary edges E_B and E_C.

Each call-return edge represents all possible control-flow paths that could be visited as a result of the call, and thus the MAY-USE, MAY-DEF, and MUST-DEF sets labeling a call-return edge indicate the register definitions and uses that occur along all of these possible paths. Initially, Spike has no information about the possible control-flow paths that could be visited during the call. Thus, each call-return edge is initialized with empty MUST-DEF, MAY-DEF, and MAY-USE sets.

3.2 Phase 1 Dataflow

After constructing the PSG, Spike performs the first phase of dataflow to find the call-used, call-defined, and call-killed sets for each routine. Figure 8 summarizes the dataflow equations used for the first dataflow phase. The call-used, call-defined, and call-killed sets summarize the

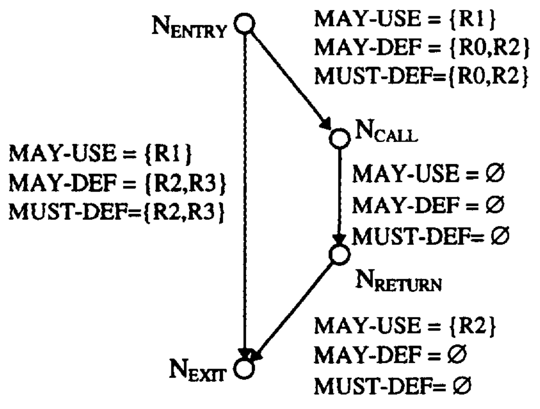

Figure 7: Complete PSG for CFG in Figure 4(a)

Initialization: for each PSG node N

$$\text{MAY-USE}[N] = \text{MAY-DEF}[N] = \text{MUST-DEF}[N] = \varnothing$$

Dataflow: for each PSG edge $E = (N_X, N_Y)$

```
MAY-USE[$N_X$] = MAY-USE[E] ∪
                 (MAY-USE[$N_Y$] - MUST-DEF[E])
MAY-DEF[$N_X$] = MAY-DEF[$N_Y$] ∪ MAY-DEF[E]
MUST-DEF[$N_X$] = MUST-DEF[$N_Y$] ∪ MUST-DEF[E]

if $N_X$ is an entry node for routine R then
    foreach call-return edge $E_{CR}$ representing a call to R
        MAY-USE[$E_{CR}$] = MAY-USE[$N_X$]
        MAY-DEF[$E_{CR}$] = MAY-DEF[$N_X$]
        MUST-DEF[$E_{CR}$] = MUST-DEF[$N_X$]
```

Figure 8: Dataflow equations to compute the call-used, call-killed, and call-defined sets for each routine.

register definitions and uses that occur as a result of a call, including definitions and uses that occur in calls made by the called routine. Thus, dataflow information flows from the entry point of the callee to the call instruction in the caller. To transmit dataflow information from a callee to a caller, Spike copies the MAY-USE, MAY-DEF, and MUST-DEF sets from an entry node to each call-return edge representing a call instruction that targets the routine represented by the entry node. When the dataflow converges, the MAY-USE, MAY-DEF, and MUST-DEF sets associated with each entry node indicate the registers call-used, call-killed, and call-defined by the routine represented by the entry node, as describe in Section 2.

Figure 9 shows the PSG for the routines in Figure 2. Dashed lines connect the call-return edges in routines P_1 and P_3 to the entry node of routine P_2, indicating the paths along which dataflow information flows from the callee to the caller. After the first phase of dataflow completes, MAY-USE[$N_{P1\text{-}ENTRY}$] = ∅, MAY-DEF[$N_{P1\text{-}ENTRY}$] = {R0,R1,R2,R3}, and MUST-DEF[$N_{P1\text{-}ENTRY}$] = {R0,R1,R2}; MAY-USE[$N_{P2\text{-}ENTRY}$] = {R1}, MAY-DEF[$N_{P2\text{-}ENTRY}$] = {R2,R3}, and MUST-DEF[$N_{P2\text{-}ENTRY}$] = {R2}; and MAY-USE[$N_{P3\text{-}ENTRY}$] = ∅, MAY-DEF[$N_{P3\text{-}ENTRY}$] = {R1,R2,R3}, and MUST-DEF[$N_{P3\text{-}ENTRY}$] = {R1,R2}. Thus, for example, for any call to routine P_1 call-used = ∅, call-defined = {R0,R1,R2}, and call-killed = {R0,R1,R2,R3}.

3.3 Phase 2 Dataflow

Spike performs the second phase of dataflow to find the live-at-entry and live-at-exit sets for each routine entry and exit. Figure 10 summarizes the dataflow equations used for the second dataflow phase. The live-at-entry and live-at-exit sets summarize the register uses that could occur along all possible returns from a routine. Thus, dataflow

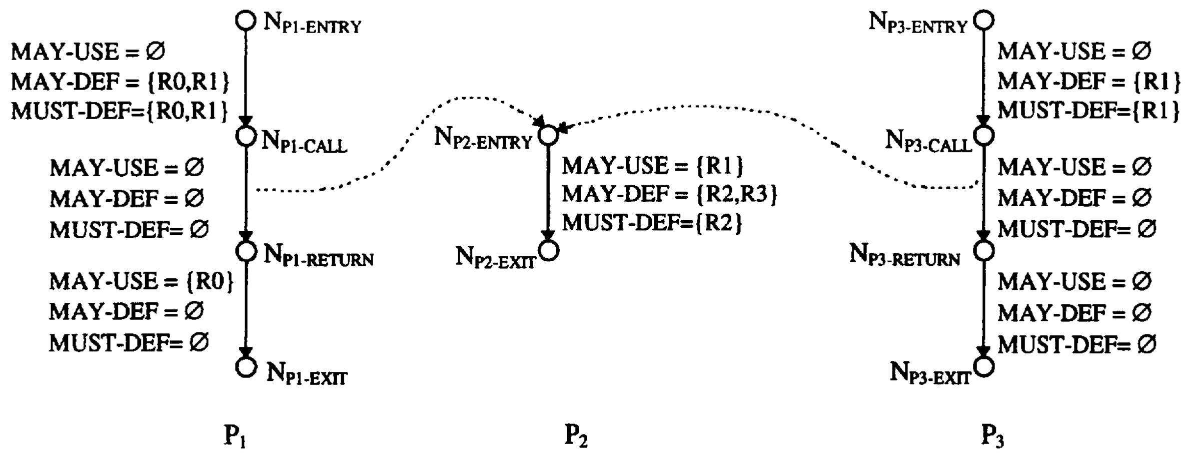

Figure 9: PSG for routines in Figure 2 before the first dataflow phase.

information flows from the return point in the caller to the exit of the callee. To transmit dataflow information from a caller to a callee, Spike copies the MAY-USE set for a return node to each exit node representing a routine exit that could return to the location represented by the return node. The MUST-DEF and MAY-USE sets computed for the call-return edges during the first dataflow phase summarize the register definitions and uses that occur during each call, and are retained for the second dataflow phase. When the dataflow converges, the MAY-USE set associated with each entry node indicates the registers live-at-entry, and the MAY-USE set associated with each exit node indicates the registers live-at-exit.

Figure 11 shows the PSG for the routines in Figure 2 before the second dataflow phase. The MAY-DEF sets are not used during the second dataflow phase and so are not shown. A dashed line connecting an exit node to a return node indicates a possible return path along which MAY-USE information flows from the caller to the callee.

```
Initialization: for each PSG node N
    MAY-USE[N] = {∅}

Dataflow: for each PSG edge E = ($N_X$, $N_Y$)
    MAY-USE[$N_X$] = MAY-USE[E] ∪
                        (MAY-USE[$N_Y$] - MUST-DEF[E])

    if $N_X$ is a return node then
        foreach exit node $N_{exit}$ representing a routine that
        could return to $N_X$
            MAY-USE[$N_{exit}$] = MAY-USE[$N_{Exit}$] ∪
                                  MAY-USE[$N_X$]
```

Figure 10: Dataflow equations to compute live-at-entry and live-at-exit sets.

3.4 Callee-Saved Registers

The Windows NT calling standard for Alpha [CALLSTD] specifies a set of *callee-saved registers*, registers that must be saved by a routine before they can be used, and that must be restored before the routine exits. As seen by the caller, a callee-saved register is not used, killed, or defined by the called routine, because the called routine saves and restores each callee-saved register it modifies. Thus, definitions and uses of callee-saved registers within a routine should not propagate to any callers of the routine, and callee-saved registers should not appear call-used, call-killed, or call-defined by any routine. During the first dataflow phase, after computing the MAY-USE, MAY-DEF, and MUST-DEF sets for an entry node, Spike removes from those sets any callee-saved registers saved and restored by the corresponding routine, preventing callee-saved register definitions and uses within a routine from propagating to the callers.

3.5 Indirect Calls and Jumps

Many programs contain indirect calls and jumps, used to implement multiway branches (e.g., for SWITCH statements in C), virtual method invocations, calls to shared library routines, etc. If Spike cannot determine the target of an indirect call or jump, conservative assumptions must be made about the registers used, defined, and killed at the target.

For multiway branches, Spike extracts the jump-table stored with the program to find all possible targets of the jump and constructs the control-flow graph using that information. If Spike cannot determine the target(s) of an indirect jump, Spike conservatively assumes that all registers are live at the jump's target. Indirect calls to an unknown target are assumed to obey the calling standard

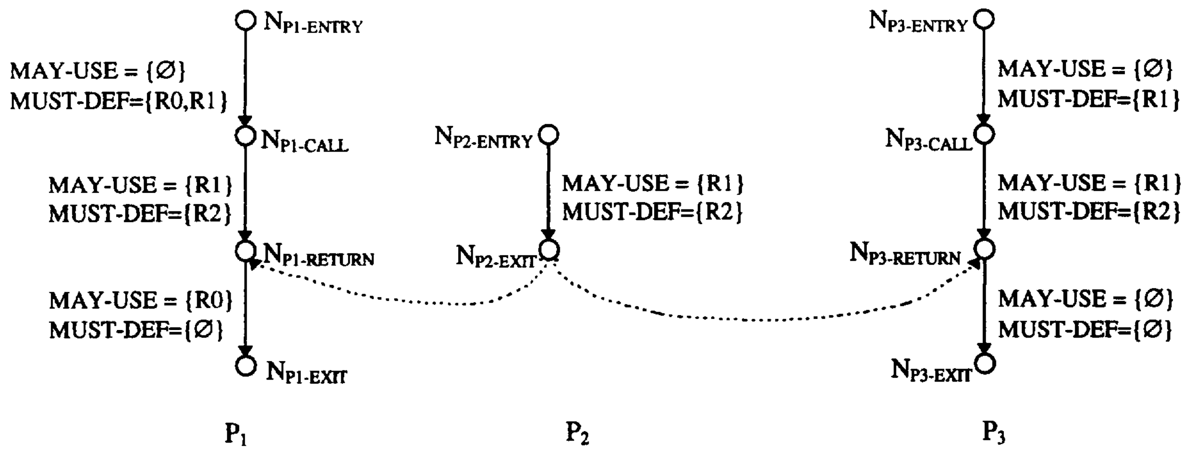

Figure 11: PSG for routines in Figure 2 before the second dataflow phase.

[CALLSTD]. Thus, Spike assumes that the registers used to pass arguments are call-used, that the return-value registers are call-defined, and that temporary registers are call-killed.

While these assumptions have proven safe for all programs optimized to date, dataflow accuracy can be improved if additional information is provided to Spike by the compiler or linker. The compiler or linker has exact information about the registers assumed to be live at the target of each indirect jump, and about the registers assumed to be call-used, call-killed, and call-defined by each indirect call. Making this information available to Spike would ensure safe and accurate dataflow information.

3.6 PSG Branch Nodes

Multiway branches inside of loops can potentially cause a large number of PSG edges to be produced for a routine, increasing the size of the PSG and thereby increasing dataflow analysis time and memory usage. To prevent a multiway branch from producing a large number of PSG edges, Spike produces a *branch node* for the multiway branch. Figure 12(a) shows the control-flow graph for a routine containing a 3-way branch with call instructions at each target of the branch. Because there is a possible control-flow path from each call instruction to itself and to every other call instruction, an edge connects every return node to every call node in the PSG representing the routine, requiring 9 flow-summary edges for the CFG in Figure 12(a). By inserting a branch node to represent the multiway branch, the number of flow-summary edges is reduced to 6, as shown in Figure 12(b). In general, inserting a branch node for an *n*-way branch potentially reduces the number of PSG edges from $O(n^2)$ to $O(n)$.

Results in Section 4 show that inserting a branch node at multiway branches can reduce the number of PSG edges by as much as 80%. A loop containing a large number of conditional (two-way) branches could potentially produce a large number of PSG edges in the same manner as a multiway branch. However, the experimental results show that inserting branch nodes for multiway branches is typically sufficient to prevent the production of a large number of PSG edges.

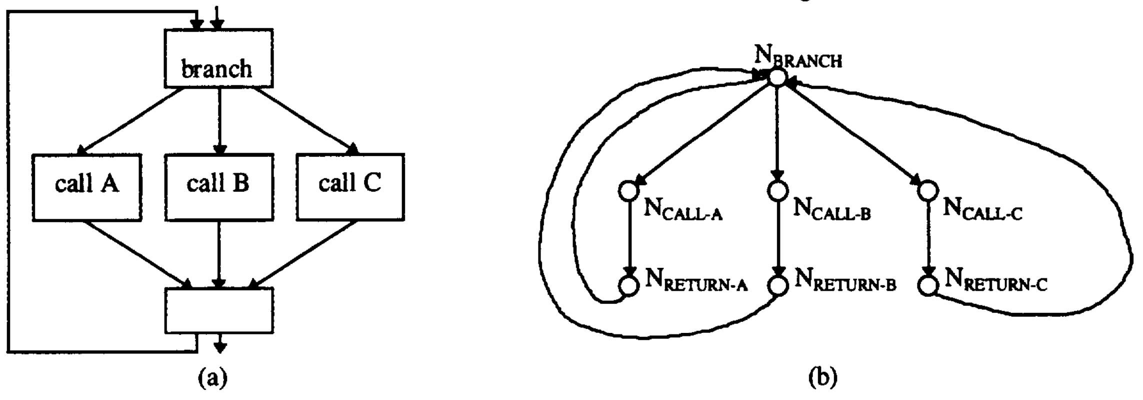

Figure 12: Example PSG edge reduction from using a branch node.

PC App	Full Name	Description
acad	Autodesk AutoCad	mechanical CAD
excel	Microsoft Excel 5.0	spreadsheet
maxeda	OrCad MaxEDA 6.0	electronic CAD
sqlservr	Microsoft Sqlservr 6.5	database
texim	Welcom Software Texim 2.0	project manager
ustation	Bentley Systems Microstation	mechanical CAD
vc	Microsoft Visual C	compiler backend
winword	Microsoft Word 6.0	word processing

Table 1: Description of each PC application benchmark.

4 Experimental Results

The interprocedural dataflow techniques presented in this paper are implemented in Spike. The techniques are evaluated using the SPEC95 integer benchmark suite and eight large PC applications compiled for Digital's Alpha architecture [ALPHA]. The PC benchmarks are described in Table 1. All results are collected on a Digital PC164LX containing a 466 Mhz 21164 processor and 128 Mbytes of memory. Each program is optimized using the same highly optimizing back-end used for Digital Unix and VMS [Blickstein92]. Table 2 shows the size of each benchmark in machine instructions, and the number of basic blocks and routines contained in the benchmark. The basic block counts assume a basic block is ended by a call instruction. With the exception of gcc, the SPEC benchmarks are dramatically smaller than the PC applications.

For each benchmark, Table 2 also shows the total time required to compute the dataflow information, and the total amount of memory required. Except for gcc, analysis time and memory usage for the SPEC integer benchmarks is insignificant. For the large benchmarks analysis time and memory usage increase, but are still reasonable given the number of instructions and basic blocks in these applications.

Figure 13 shows the fraction of the total dataflow time spent in different stages of the analysis. The smaller benchmarks are not shown because the timer resolution prevented accurate timing of each phase. *CFG Build* indicates the fraction of time spent building the CFG for each routine. *Initialization* indicates the fraction of time spent in general initialization and consists mainly of the time spent generating the DEF and UBD sets for each basic block. *PSG Build* indicates the fraction of time spent generating the PSG nodes and edges. *Phase 1* and *Phase 2* indicate the fraction of time spent in the two phases of dataflow analysis. For each benchmark the fraction of time spent in initialization and control-flow graph building is consistently 50-60% of the total analysis time. However, the fraction of time spent in the remaining stages varies considerably between the benchmarks.

Table 3 shows several benchmark characteristics that influence the time required to construct the PSG nodes and edges for each routine, and the number of PSG nodes and PSG edges needed for each routine. The number of calls in a routine and the number of entrances and exits to and from a routine, determine the number of PSG nodes required for the routine. For example, maxeda has a large number of calls per routine on average, and thus requires a

Suite	Benchmark	Routines	Basic Blocks	Instructions (k)	Total Dataflow Time (sec.)	Memory Usage (Mbytes)
SPECint95	compress	122	2546	13.5	0.05	.20
	gcc	1878	69588	297.6	1.90	6.38
	go	462	12548	71.4	0.28	.88
	ijpeg	393	6814	42.8	0.16	.56
	li	491	6052	29.4	0.14	.56
	m88ksim	383	8205	40.6	0.16	.58
	perl	487	19468	92.7	0.42	1.57
	vortex	818	21880	110.0	0.59	2.85
PC Applications	acad	31766	339962	1734.7	12.04	41.11
	excel	12657	301823	1506.3	8.95	28.04
	maxeda	2126	84053	418.6	2.02	8.14
	sqlservr	3275	123607	754.9	3.34	10.17
	texim	1821	50955	302.0	1.34	5.36
	ustation	12101	165929	916.4	5.21	16.61
	vc	2154	82072	493.7	2.18	6.18
	winword	12252	288799	1520.8	8.30	25.42

Table 2: Benchmark size, dataflow analysis time and memory usage.

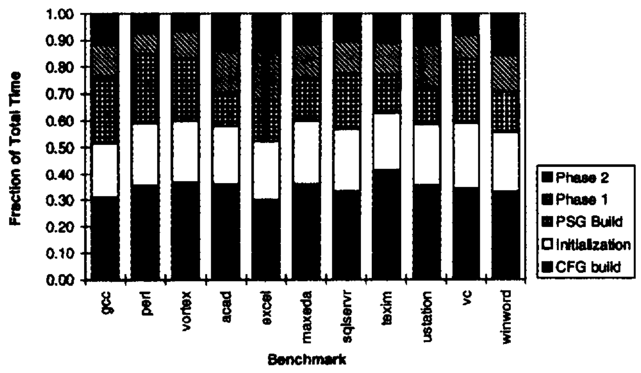

Figure 13: Fraction of total time spent in different stages of the dataflow analysis.

large number of PSG nodes for each routine.

A routine containing a large number of branches is more likely to have multiple control-flow paths connecting calls, entrances, and exits, and thus is likely to require more PSG edges than a routine containing fewer branches. Moreover, to compute the MUST-DEF, MAY-DEF, and MAY-USE sets for each PSG edge, Spike must construct and analyze a subgraph of the CFG, increasing the time required to build the PSG. For example, because `acad` has significantly fewer PSG edges per routine than `vc`, `acad` spends a smaller fraction of the total analysis time building the PSG than `vc`.

Branch nodes significantly reduce the number of PSG edges required to represent a routine. Table 4 shows the reduction for each benchmark, compared with a PSG constructed without branch nodes. For several benchmarks, branch nodes dramatically reduce the number of PSG edges, while increasing the number of PSG nodes by less than 1%.

Using the PSG is an effective technique for reducing the size of the graph over which dataflow must be computed. For each benchmark, Table 5 shows the number of nodes and edges in the PSG, and the number of basic blocks and control-flow arcs in the CFG, including arcs representing calls and returns. On average the PSG has 30% fewer nodes than the CFG has basic blocks, and has nearly 40% fewer edges than the CFG has arcs. Two benchmarks, `acad` and `vortex`, show uncharacteristic results. `acad` has a high density of call instructions, where on average there is a call every 2.1 basic blocks. Each call instruction requires two PSG nodes, which along with the entry and exit nodes causes `acad` to have more PSG nodes than basic blocks. The PSG for `vortex` contains more edges than there are arcs in the CFG due to the large number of branches inside loops, as discussed in Section 3.6.

Typically, the PSG for a program contains significantly fewer nodes than the CFG contains basic blocks. Moreover, each PSG node occupies significantly less memory than a basic block. When using the CFG for interprocedural dataflow analysis, the amount of dataflow information that must be maintained in each basic block is approximately equal to the dataflow information contained in three PSG nodes. Each basic block contains the MAY-USE$_{IN}$, MAY-USE$_{OUT}$, MAY-DEF$_{IN}$, MAY-DEF$_{OUT}$, MUST-DEF$_{IN}$, and MUST-DEF$_{OUT}$ dataflow sets as well as the DEF and UBD sets indicating which registers are defined and used-before-defined in the basic block. In contrast, a PSG node contains just three dataflow sets,

Suite	Benchmark	Entrances/ Routine	Exits/ Routine	Calls/ Routine	Branches/ Routine	PSG Nodes/ Routine	PSG Edges/ Routine
SPECint95	compress	1.04	1.81	3.30	13.75	9.47	17.19
	gcc	1.00	1.62	9.86	23.16	22.45	43.65
	go	1.01	1.71	4.92	17.99	12.58	22.03
	ijpeg	1.02	1.49	3.92	10.55	10.38	16.16
	li	1.01	1.37	3.49	7.18	9.41	10.72
	m88ksim	1.02	1.75	4.66	13.47	12.14	16.39
	perl	1.01	1.47	9.34	25.55	21.27	40.73
	vortex	1.01	1.20	8.97	15.00	20.19	50.11
PC Applications	acad	1.00	1.14	5.02	4.58	12.18	14.36
	excel	1.00	1.00	8.42	12.98	18.88	26.66
	maxeda	1.00	1.12	15.45	20.25	32.96	46.33
	sqlservr	1.02	1.30	10.48	22.60	23.31	38.94
	texim	1.00	1.29	11.24	13.90	24.91	34.47
	ustation	1.00	1.35	5.03	6.86	12.42	15.76
	vc	1.03	1.10	9.11	24.47	20.51	36.58
	winword	1.00	1.01	8.10	13.02	18.25	24.64

Table 3: Benchmark characteristics influencing PSG size and construction time.

Benchmark	PSG Edge Reduction	PSG Node Increase
compress	35.4%	0.4%
gcc	48.5%	0.5%
go	12.2%	0.2%
ijpeg	17.1%	0.2%
li	1.3%	0.4%
m88ksim	1.2%	0.5%
perl	73.6%	0.5%
vortex	4.7%	0.2%
acad	1.8%	0.2%
excel	4.1%	0.4%
maxeda	0.9%	0.3%
sqlservr	80.0%	0.2%
texim	3.6%	0.6%
ustation	2.1%	0.2%
vc	55.4%	0.8%
winword	0.3%	0.3%

Table 4: PSG edge reduction provided by branch nodes.

MAY-USE, MAY-DEF, and MUST-DEF.

Figure 14 shows the total time required to perform the interprocedural dataflow analysis for each benchmark, as a function of the number of instructions, as a function of the number of basic blocks, and as a function of the number of routines. The analysis time is well behaved, especially with respect to the number of basic blocks, and shows low-order polynomial complexity. Figure 15 shows the memory required to perform the interprocedural dataflow analysis for each benchmark. As with analysis time, the memory required is well behaved and has low-order polynomial complexity.

5 Related Work

Spike uses the two-phase dataflow approach proposed in [Srivastava93] to precisely compute the meet-over-all-valid-paths solution [Callahan88, Reps95, Sagiv95, Sharir81] for the live-at-entry and live-at-exit dataflow sets. A path through a called routine is valid if it returns to the site of the call. The first dataflow phase computes the registers defined and used by a call to each routine, not including any register use that occurs after a routine returns. During the second dataflow phase, the registers live before a call instruction are influenced only by the registers used and defined during the call and by the registers live after that call instruction, but not by registers that are live at the called routine's other return sites. Thus, the live-at-entry and live-at-exit sets computed during the second dataflow phase include only the registers that are live over valid paths.

While the program summary graph used by Spike is conceptually similar to the one proposed by Callahan [Callahan88], it differs significantly in its construction and in its encoded information. Callahan's PSG contains nodes and edges for each variable being analyzed, while Spike's PSG contains a single set of nodes and edges shared by all the registers. For each variable, Callahan builds the PSG to represent the portion of the CFG reachable by definitions of the variable. Callahan's PSG edges do not encode any variable definition or use information. Spike, on the other hand, builds the PSG to represent an entire routine and summarizes on the edges the register definitions and uses that occur on the control-flow paths represented by the edge.

Suite	Benchmark	PSG Nodes (k)	PSG Edges (k)	Basic Blocks (k)	CFG Arcs (k)	Nodes / Basic Block	Edges / Arc
SPECint95	compress	1.16	2.10	2.55	4.20	.45	.50
	gcc	42.16	81.97	69.59	125.91	.61	.65
	go	5.81	10.18	12.55	21.95	.46	.46
	ijpeg	4.08	6.35	6.81	11.39	.60	.56
	li	4.62	5.27	6.05	10.74	.76	.49
	m88ksim	4.65	6.28	8.21	14.02	.57	.45
	perl	10.36	19.84	19.47	33.72	.53	.59
	vortex	16.51	40.99	21.88	39.95	.75	1.03
PC Applications	acad	386.80	456.07	339.96	612.11	1.14	.75
	excel	238.91	337.48	301.82	544.41	.80	.62
	maxeda	70.08	98.50	84.05	151.55	.83	.65
	sqlservr	76.33	127.54	123.61	211.74	.62	.60
	texim	45.36	62.77	50.96	90.79	.89	.69
	ustation	150.27	190.76	165.93	294.47	.91	.65
	vc	44.17	78.80	82.07	146.34	.54	.54
	winword	223.56	301.84	288.80	508.20	.77	.59

Table 5: Comparison of PSG nodes and edges to CFG basic blocks and arcs.

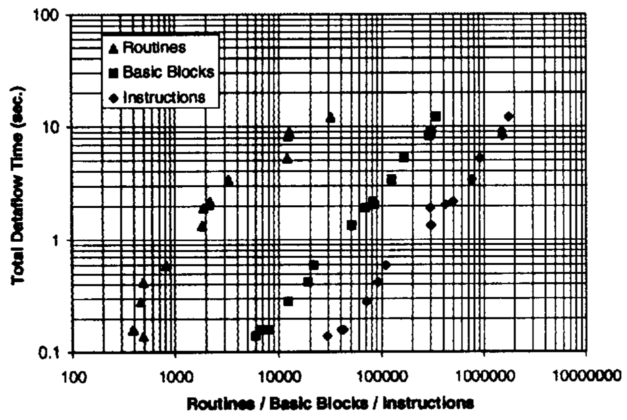

Figure 14: Total interprocedural dataflow analysis time for each benchmark as a function of number of routines, basic blocks, and instructions in the benchmark.

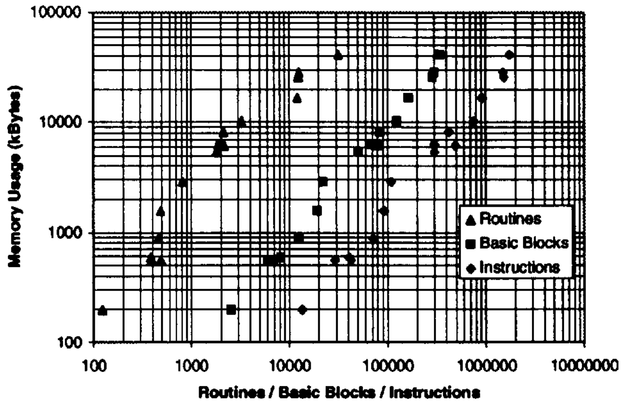

Figure 15: Memory usage for each benchmark as a function of number of routines, basic blocks, and instructions in the benchmark.

The branch nodes described in Section 3.6 can potentially allow a quadratic number of PSG edges to be replaced with a linear number of edges. Thus, branch nodes have similarities to SSA form [Cytron89] and to earlier techniques [Reif82] that reduce the number of def-use chains used to connect variable definitions and uses.

6 Conclusions

The Spike executable optimizer uses interprocedural dataflow information to enable analyses and code transformations that are not possible in a traditional compiler. Computing interprocedural dataflow information for large programs could potentially require unacceptable amounts of time and memory, severely restricting the applicability and effectiveness of Spike's optimizations. The techniques and results presented in this paper demonstrate that interprocedural dataflow analysis is practical even for large applications containing millions of machine instructions, and hundreds of thousands of basic blocks.

To perform interprocedural dataflow analysis, Spike first constructs a program summary graph to represent the program's intraprocedural and interprocedural control flow, and to represent the register definitions and uses that occur in the program. Next, Spike uses traditional dataflow techniques to compute the sets of registers live at each routine entry and exit, and the sets of registers used, killed, and defined by each call. These dataflow sets provide a summary of the register definitions, uses, and kills that occur external to each routine, allowing Spike to perform optimizations across call instructions and procedure boundaries.

Spike's interprocedural dataflow analysis is evaluated using the SPEC95 integer benchmarks and eight large PC applications. For the small benchmarks, interprocedural dataflow analysis uses an insignificant amount of time, requiring less than two seconds for each of the SPEC95 integer benchmarks. For the larger benchmarks, analysis time and memory usage increase as a near-linear function of program size, remaining practical even for the largest programs.

7 Acknowledgments

The author would like to thank Robert Cohn and Geoff Lowney for their help implementing and testing interprocedural dataflow analysis in Spike.

8 References

[Aho88] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1988.

[ALPHA] Alpha Architecture Reference Manual. http://www.partner.digital.com/www-swdev/pages/Home/TECH/documents/alpha_cookbook/biblio.htm

[Blickstein92] D. Blickstein, et al, "The GEM optimizing compiler system," *Digital Technical Journal*, 4(4):121-136.

[Callahan88] D. Callahan. "The program summary graph and flow-sensitive interprocedural data flow analysis," in *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation '88*, pp 47-56, June 1988.

[CALLSTD] Alpha NT Calling Standard. http://www.partner.digital.com/www-swdev/pages/Home/TECH/documents/alpha_cookbook/biblio.htm

[Cohn96] R. Cohn and G. Lowney, "Hot Cold optimization of large Windows/NT applications," to appear in *Proc. of the 29th Annual Intl. Symp. on Microarchitecture*, Dec. 1996.

[Cytron89] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman, and Kenneth Zadeck, "An efficient method for computing static single assignment form." in *ACM Symposium on Principles of Programming Languages*, pp 25-35, January 1989

[Goodwin96] D. Goodwin and K. Wilken, "Optimal and near-optimal global register allocation using 0-1 integer programming," *Software - Practice & Experience*, Vol. 26(8), pp. 929-965, August 1996.

[Khedker94] U. Khedker and D. Dhamdhere, "A generalized theory of bit vector data flow analysis," *ACM Transactions on Programming Languages and Systems*, Vol. 16(5), pp. 1472-1511, September 1994.

[Pettis90] K. Pettis and R. Hansen, "Profile guided code positioning," in *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation '90*, pp 16-27, June 1990

[Reif82] John Reif and Robert Tarjan, "Symbolic program analysis in almost linear time." *SIAM Journal of Computing*, 11(1), February 1982.

[Reps95] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proc. 22nd ACM SIGPLAN-SIGACT Conf. on Principles of Programming Languages*, pp. 49-61, 1995.

[Sagiv95] M. Sagiv, T. Reps, and S. Horowitz, "Precise interprocedural dataflow analysis with applications to constant propagation," in *Proc. of FASE '95: Colloquium on Formal Approaches in Software Engineering*, May 1995. *Lecture Notes in Computer Science*, Vol. 915, Springer-Verlag, New York, NY, pp. 651-665.

[Sharir81] M. Sharir and A. Pnueli, "Two approaches to interprocedural data flow analysis." in *Program Flow Analysis, Theory and Applications* by S. Muchnick and N. Jones, 1981.

[Srivastava93] A. Srivastava and D. Wall, "A practical system for intermodule code optimization at link-time," *Journal of Programming Languages 1(1)*, pp. 1-18, March 1993.

[Srivastava94] A. Srivastava and D. Wall, "Link-time optimization of address calculation on a 64-bit architecture," *in Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation '94*, pp. 49-60, Orlando, FL, June 1994.

[Wilson96] L.S. Wilson, C.A. Neth, M.J. Rickabaugh, "Delivering binary object modification tools for program analysis and optimization," volume 8,1 of *Digital Technical Journal*, pp. 18-31, 1996.