The Journal of Systems and Software 146 (2018) 186–195

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Parallel construction of interprocedural memory SSA form

Yulei Sui[a,*], Hua Yan[b], Zheng Zheng[c], Yunpeng Zhang[d], Jingling Xue[b]

[a] University of Technology Sydney, Australia
[b] The University of New South Wales, Australia
[c] Beihang University, China
[d] University of Houston, TX, United States

ARTICLE INFO

Article history:
Received 22 March 2018
Revised 3 August 2018
Accepted 12 September 2018
Available online 13 September 2018

ABSTRACT

Interprocedural memory SSA form, which provides a sparse data-flow representation for indirect memory operations, paves the way for many advanced program analyses. Any performance improvement for memory SSA construction benefits for a wide range of clients (e.g., bug detection and compiler optimisations). However, its construction is much more expensive than that for scalar-based SSA form. The memory objects distinguished at a pointer dereference significantly increases the number of variables that need to be put on SSA form, resulting in considerable analysis overhead when analyzing large programs (e.g., millions of lines of code).

This paper presents PARSSA, a fully parameterised approach for parallel construction of interprocedural memory SSA form by utilising multi-core computing resources. PARSSA partitions whole-program memory objects into uniquely identified memory regions. The indirect memory accesses in a function are fully parameterised using partitioned memory regions, so that the memory SSA construction of a parameterised function is readily parallelised. We implemented PARSSA in LLVM using Intel Threading Building Block (TBB) for creating parallel tasks. We evaluated PARSSA using 15 large applications. PARSSA achieves up to 6.9× speedup against the sequential version on an 8-core machine.

1. Introduction

Static Single Assignment (SSA) form (Rosen et al., 1988) is the mainstream intermediate representation used to perform analyses and optimizations of scalars in modern compilers (e.g., LLVM (Lattner and Adve, 2014), GCC (Novillo and Canada, 2007), and Java Hotspot (Kotzmann et al., 2008)). It provides a sparse data-flow representation in which every variable can only be defined once. To enable the sparsity of both scalars and indirect memory operations, various memory SSA forms (e.g., factored SSA (Choi et al., 1994), HSSA (Chow et al., 1996), and Tree SSA (Novillo and Canada, 2007)) have been proposed to support aggressive compiler optimizations. To reduce compile-time overhead, majority of their construction algorithms are *intraprocedural*, i.e, a pair of pointer dereferences in a function *f* are conservatively treated as aliases if both may access memory objects defined outside *f*.

Compared to light-weight intraprocedural memory SSA form, its *interprocedural* counterpart provides fine-grained memory dependence by considering global alias information across functions. Due to improved precision, the resulting SSA representation is useful for many client applications, such as flow-sensitive pointer analysis (Hardekopf and Lin, 2011; Sui and Xue, 2016a), static memory error detection (Livshits and Lam, 2003; Sui et al., 2012), change impact analysis (Guo et al., 2016; Cai et al., 2016) and identifying redundant instrumentations to accelerate dynamic analysis (Ye et al., 2014).

Constructing interprocedural memory SSA form is expensive. Because of the undecidability of aliases (Landi, 1992), a memory operation (load or store) may access many different memory objects at a pointer dereference due to over-approximation. Unlike intraprocedural memory SSA forms in Open64 (Chow et al., 1996) and GCC (Novillo and Canada, 2007), which use a single *virtual symbol* (Chow et al., 1996) to represent all memory objects defined outside a function, an interprocedural SSA form distinguishes every object at a memory access with a unique name for SSA renaming, resulting in precise dependences between two memory operations. However, distinguishing objects accessed at pointer dereferences significantly increases the number of variables that need to be put in SSA. A memory SSA construction algorithm which involves a non-trivial data flow analysis (Novillo and Canada, 2007) takes substantial time for analysing large programs.

* Corresponding author.
E-mail address: yulei.sui@uts.edu.au (Y. Sui).

https://doi.org/10.1016/j.jss.2018.09.038
0164-1212/

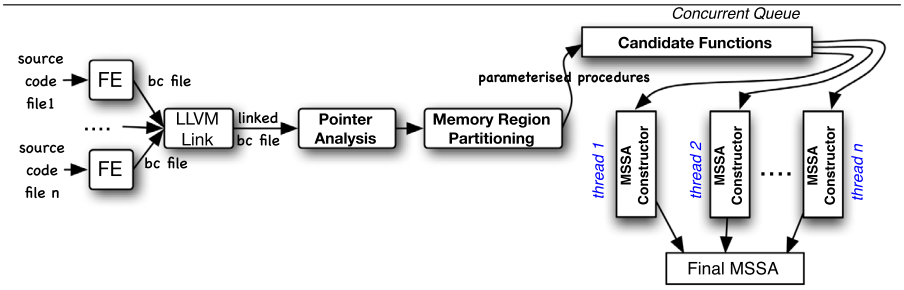


Fig. 1. ParSSA workflow.

Nowadays, multi-core platforms are ubiquitous. It becomes imperative to exploit parallelism to accelerate memory SSA construction algorithms. However, such algorithms are often not directly amenable to parallelisation. For example, the algorithm in Hardekopf and Lin (2011) works on the interprocedural control-flow graph (ICFG (Landi and Ryder, 1992)) of a program. The approach is designed to be entirely sequential, by treating the whole program as a single graph, thereby hindering its parallelisation.

In this paper, we present ParSSA, a simple yet effective **Par**allel approach to accelerating the construction of interprocedural memory **SSA**. ParSSA enables pre-analysis to partition the whole-program abstract memory objects into memory regions. Every region is uniquely identified in a program to represent a set of objects that are accessed equivalently using the results from an existing pointer analysis. The interprocedural memory dependences of a function are then fully parameterised using memory regions, so that the memory SSA construction of individual functions can be readily parallelised. The key contributions of this paper are:

- We propose ParSSA, the first parallel approach to constructing interprocedural memory SSA form for large-scale programs with millions of lines of code.
- We present a new approach to constructing fully parameterised interprocedural memory SSA form using memory region partitioning. The source code is available at https://github.com/SVF-tools/SVF/tree/master/lib/MSSA.
- We have evaluated ParSSA using a set of 15 large applications. ParSSA achieves up to 6.9× speedup against the sequential version on an 8-core machine.

The rest of this paper is organised as follows. Section 2 presents our ParSSA approach including the overview of ParSSA framework (Section 2.1), the examples of intraprocedural and interprocedural memory SSA forms (Section 2.2.1), the side-effect analysis (Section 2.2.2) and memory region generation and parallel construction (Section 2.2.3). Section 3 evaluates ParSSA including implementation (Section 3.3), methodology (Section 3.2), results and analysis (Section 3.3). Section 4 describes the related work. Finally, Section 5 concludes the paper and discusses some future work.

2. ParSSA approach

The key idea of our ParSSA approach is to parameterise every function of a program through partitioned memory regions, so that the indirect memory accesses in a function are fully parameterised through the side-effect analysis using these memory regions. Therefore, memory dependences across functions are decoupled, making memory SSA form construction readily parallelised.

2.1. Overview of ParSSA

The workflow of ParSSA is depicted in Fig. 1. The source code of a program is first compiled by the clang compiler front-end (FE) into bit-code files, which are merged by LLVM Gold Plugin (llv) at link time stage to produce a whole-program bit-code file. Then the "Pointer Analysis" module is invoked. Based on the points-to information obtained, we first perform a lightweight side-effect analysis to capture interprocedural reference and modification of each abstract memory object. Thus, the (alias) set of indirect defs (uses) at a statement ℓ (i.e., a store, load or callsite) in each function is obtained and denoted as $\mathbb{D}_\ell$ ($\mathbb{U}_\ell$).

The "Mem Region Partitioning" module partitions all the abstract memory objects of a program into a set of disjointed regions $R_1, \ldots, R_n$. Then every statement ℓ is annotated with each R_i, where $\mathbb{D}_\ell \cap R_i \neq \varnothing$ ($\mathbb{U}_\ell \cap R_i \neq \varnothing$), to make explicit the memory objects that may be defined (used) indirectly at ℓ. Once indirect uses and defs are identified, the interprocedural dependences are fully parameterised for every function by using uniquely named regions, so that we can achieve function level parallelism to produce a whole-program memory SSA form that has the same precision as the one built by a sequential algorithm. Our algorithm for constructing fully parameterised SSA form has been implemented in the open-source tool SVF (Sui and Xue, 2016b) (https://github.com/SVF-tools/SVF) based on the LLVM compiler.

ParSSA uses Intel Threading Building Block (TBB) to fork multiple threads for building SSA form for each parameterised functions. The *concurrent queue* data structure is used to store all the parameterised functions of a program after memory region partitioning. ParSSA performs parallel construction of memory SSA for every program function by allocating parallel tasks ("MSSA constructor") using *task groups* in TBB, so that every allocated task constructs memory SSA modularly by choosing the next available parameterised function from the concurrent queue. Finally, the whole program memory SSA form is available when all the parallel tasks finish.

2.2. Parameterised memory SSA form

This section details our ParSSA approach. Section 2.2.1 describes the background knowledge and examples of memory SSA forms. Section 2.2.2 introduces whole-program side-effect analysis to discover interprocedural dependences across functions using results from a pointer analysis. Based on the side-effect analysis, Section 2.2.3 discusses memory region generation to parameterise program functions to enable parallel memory SSA construction.

2.2.1. Intraprocedural and interprocedural Memory SSA Form Examples

Without loss of generality, we follow the LLVM convention (Hardekopf and Lin, 2011; Sui and Xue, 2016a; Lhoták and Chung, 2011) of separating all variables in a program into two disjoint sets: $\mathcal{A}$ containing all possible targets, i.e., *address-taken variables* of a pointer and $\mathcal{T}$ containing all *top-level variables*.

A program is represented by five types of statements: $p{=}\&a$ (AddrOf), $p{=}q$ (Copy), $p{=}{*}q$ (Load), ${*}p{=}q$ (Store), and $p{=}\phi(q,r)$

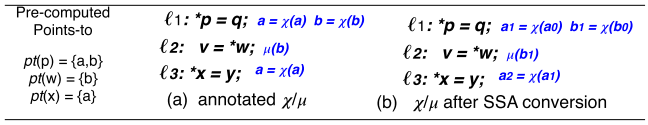


Fig. 2. Intraprocedural memory SSA with μ/χ at stores/loads.

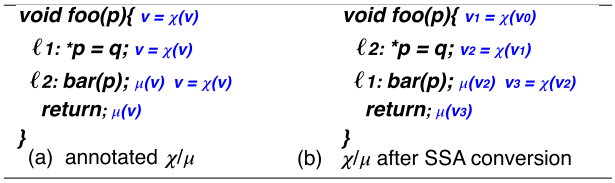


Fig. 3. Interprocedural memory SSA with μ/χ at callsite and function entry/exit.

(Phi), where $p, q, r \in \mathcal{T}$ and $a \in \mathcal{A}$. For an AddrOf statement $p{=}\&a$, known as an *allocation site*, a is a stack or global variable with its address taken or a dynamically created abstract heap object (at, e.g., a `malloc()` site). Interprocedural parameter assignments and function returns are modeled using Copy.

Top-level variables can be put directly in SSA form using standard SSA construction algorithm (e.g., Cytron et al., 1991) without requiring any pointer analysis. Address-taken variables are only accessed indirectly via Load or Store. Each of them can be indirectly defined multiple times which requires pointer analysis to discover their defs and uses, thus they are more complicated for SSA conversion.

In order to explicitly put address-taken variables on SSA, we adopt the approach in Chow et al. (1996) by introducing μ and χ operators to represent these possible uses and defs. As illustrated in Fig. 2(a), each indirect store (e.g., $*p = q$) in the original program is annotated with an operator $a = \chi(a)$ to represent a potential def and use of a at the store based on its pre-computed points-to information. If a can be strongly updated, then a receives whatever q points to and the old contents in a are killed. Otherwise, a must also incorporate its old contents, resulting in a weak update to a. Similarly, each indirect load (e.g., $v = *w$) in the original program is annotated with an operator $\mu(b)$ for each variable b that may be accessed by the load. Finally, each address-taken variable, e.g., b is converted into SSA form (Fig. 2(b)), with each $\mu(b)$ treated as a use of b, and each $b = \chi(b)$ as both a def and use of b.

When considering function calls, the memory SSA is more complicated to construct. To build per-function SSA, the previous intraprocedural approaches (Chow et al., 1996; Novillo and Canada, 2007) does not analyse the side-effect of a function call. Instead, it conservatively uses a single virtual variable v in every function f to represent all the non-local objects in f. An object represented by v is assumed to be modified (read) at any store and callsite (load and callsite) in f. As illustrated in Fig. 3(a), both store ℓ_1 and callsite ℓ_2 are annotated with $v = \chi(v)$ to capture defs of all non-local objects in foo. Likewise, $v = \chi(v)$ ($\mu(v)$) is annotated at the entry (exit) of foo to mimic the parameter passing (return). However, using v to represent all non-local objects are overly conservative, which may produce overwhelming spurious dependences, e.g., an object defined at ℓ_1 is always assumed to be modified via callsite at ℓ_2, even if there is no store statement via dereference $*p$ in callee bar.

A fine-grained solution is to build a single SSA form over the whole-program ICFG (Landi and Ryder, 1992) using points-to results. However, such approach makes SSA construction inefficient when the size of a program grows. Moreover, it makes parallel construction impossible due to densely coupled dependences across the functions.

2.2.2. Whole-program side-effect analysis

This section introduces our side-effect analysis to discover interprocedural program dependences of a function using the results of Andersen's pointer analysis (Andersen, 1994). Given a function f, the side-effect analysis determines the set $\mathbb{U}$ ($\mathbb{D}$) of the nonlocal memory objects (Definition 1) in f that may be indirectly read (modified) when f is executed, denoted as $f : \mathbb{U}, \mathbb{D}$. The side-effect of each statement $\ell \in \mathbb{L}_f$ in a function f, denoted as $\ell : \mathbb{U}, \mathbb{D}$, is analysed individually.

Fig. 4 gives the rules of our side-effect analysis. The root causes for the interprocedural side-effect are loads and stores. For a load $p = *q$, the points-to set $pt(q)$ of q may contain nonlocal objects read in f (`[LOAD]`). Similarly, `[STORE]` collects nonlocal objects in $pt(q)$ that may be modified at a store. In contrast, address and copy statements do not contribute any side-effect according to `[ALLOC]` and `[COPY]`. Rule `[PROC]` simply collects the side-effect of a function f by accumulating the computed side-effect of its statements.

For a callsite $\ell : \_ = f(\_)$ with its callee function f, the most conservative side-effect analysis is to assume that the set of all variables passed into this callsite may be read and modified by its callees invoked directly/indirectly. This naive approach is inaccurate due to a large number of unrealisable def-use chains created across the functions. Therefore, we only collect objects $E_{\ell \to f}$ (Definition 2), which are escaped from callsite ℓ to its callee f as computed based on lines 6-10 in Algorithm 1. In the presence of recursion, `[CALL]` and `[PROC]` are recursively applied until a fixed point is reached.

Definition 1 (Nonlocal Objects)**.** Consider a memory object $o \in \mathcal{A}$ that is not a global object but accessed in a function f. We say that o represents a *local* object if (1) o is locally declared in f and (2) f does not appear in any recursion cycle, and a *nonlocal* object otherwise. We write $Local_f$ ($NonLocal_f$) to represent the set of all local (nonlocal) objects accessed in f.

Definition 2 (Callsite Escaped Objects)**.** For a callsite $\ell : \_ = f(\_)$ with its callee function f, a set of escaped objects $E_{\ell \to f}$ represents all nonlocal objects passed into callsite ℓ that may be used or modified inside callee function f. $E_{\ell \to f}$ is pre-computed using Andersen's points-to results according to Algorithm 1.

Theorem 1. *(Soundness).* ***Proof Sketch:*** *Our side-effect analysis is sound because (1) the side-effect of a statement $\ell : \mathbb{U}, \mathbb{D}$ (`[LOAD]`, `[STORE]` and `[CALL]`) is over-approximated due to the underlying sound pointer analysis, and (2) $f : \mathbb{U}, \mathbb{D}$ records all the nonlocal locations in $NonLocal_f$ read and modified by f (`[PROC]`).*

2.2.3. Memory region generation

After side-effect analysis, we generate a set of memory regions, denoted Υ. Every memory region represents a set of memory objects. Any two memory regions R, $R' \in \Upsilon$ are disjointed, i.e., $R \cap R' = \Upsilon$. Algorithm 2 describes the region generation for function f. Initially, memory regions are collected from pointer dereferences based on points-to information (lines 2-3) and callsites based on side-effect analysis (lines 4-5). Then the regions are gradually refined by making all regions disjointed (lines 7-10) using a standard worklist algorithm.

Note that memory region generation is not limited to the results of a particular pointer analysis. More precise points-to information can help generate regions that have more precise dependence relations for memory SSA construction.

2.2.4. Parallel construction

Algorithm 3 describes the sequential version of constructing the memory SSA form for a parameterised function. There are three

$$[\texttt{ALLOC}]\ \frac{\ell : p = \&a}{\ell : \varnothing; \varnothing} \qquad [\texttt{COPY}]\ \frac{\ell : p = q}{\ell : \varnothing; \varnothing}$$

$$[\texttt{LOAD}]\ \frac{\ell : p = *q \quad \ell \in \mathbb{L}_f \quad \mathbb{U} = \{a \mid a \in pt(q) \wedge a \in NonLocal_f\}}{\ell : \mathbb{U}; \varnothing} \qquad [\texttt{STORE}]\ \frac{\ell : *p = q \quad \ell \in \mathbb{L}_f \quad \mathbb{D} = \{a \mid a \in pt(p) \wedge a \in NonLocal_f\}}{\ell : \varnothing; \mathbb{D}}$$

$$[\texttt{PROC}]\ \frac{\{\ell_1 \ldots \ell_n\} \subseteq \mathbb{L}_f \quad \ell_i : \mathbb{U}_i; \mathbb{D}_i \quad i \in [1, n]}{f : \bigcup_i \mathbb{U}_i; \bigcup_i \mathbb{D}_i} \qquad [\texttt{CALL}]\ \frac{\ell : \_ = f(\_) \quad f : \mathbb{U}; \mathbb{D}}{\ell : \mathbb{U} \cap E_{\ell \to f}; \mathbb{D} \cap E_{\ell \to f}}$$

Fig. 4. Rules to determine side-effect of a function including loads/stores and callsites.

Function ComputeEscapedObjs($\ell : \_ = f(p, ...)$)
begin
1 $W \leftarrow \varnothing$; $\mathcal{O} \leftarrow \varnothing$;
2 **foreach** *parameter* p *at callsite* $\ell : \_ = f(p, ...)$ **do**
3 $W \leftarrow W \cup pt(p)$;
4 **foreach** *global pointer* g **do**
5 $W \leftarrow W \cup pt(g)$;
6 **while** $W \neq \varnothing$ **do**
7 $o = W.pop()$;
8 **if** $\mathcal{O} \cap \{o\} = \varnothing$ **then**
9 $\mathcal{O} = \mathcal{O} \cup \{o\}$; $W = W \cup pt(o)$;
$E_{\ell \to f} = \{o \mid o \in \mathcal{O} \wedge o$ is non local object$\}$;
10 **return** $E_{\ell \to f}$;

Algorithm 1. Computing escaped objects

Function GenerateRegion(f)
begin
1 $W \leftarrow \varnothing$;
2 **foreach** *pointer dereference* $*p$ *in function* f **do**
3 $R \leftarrow pt(p)$; $W \leftarrow W \cup \{R\}$;
4 **foreach** *callsite* $\ell : \_f(\_) : \mathbb{U}; \mathbb{D}$ *in* f **do**
5 $R \leftarrow \mathbb{U}$; $R' \leftarrow \mathbb{D}$; $W \leftarrow W \cup \{R\} \cup \{R'\}$;
6 $changed = true$;
7 **while** *changed* **do**
8 **if** $\exists R_1, R_2 \in W : R_1 \cap R_2 \neq \varnothing$ **then**
9 $R_3 \leftarrow R_1 \cap R_2$; $R_1' \leftarrow R_1 \backslash R_3$; $R_2' \leftarrow R_2 \backslash R_3$;
10 $W \leftarrow W \cup \{R_1'\} \cup \{R_2'\} \cup \{R_3\}$; $W \leftarrow W \backslash \{R_1, R_2\}$;
11 **else** $changed = false$;
12 **return** W;

Algorithm 2. Memory region generation

phases (lines 1–3): (1) creating μ and χ annotations for memory regions (lines 4–9); (2) adding ϕ functions for multiple definitions of the same region that are live at join points of control flows (lines 10–18), and (3) performing SSA conversion to rename all instances of regions (lines 19–36).

Algorithm 4 performs parallel construction for each function using the allocated TBB task. Each task forks a thread executing Algorithm 3. The algorithm starts with the ParallelConstruct method with *N* threads (lines 1-3). Each task selects a parameterised function *f* from the shared concurrent queue for building *f*'s memory SSA form. Note that there is no need for synchronisation among the three phases of Algorithm 3 since every function is parameterised using globally partitioned memory regions.

Theorem 2. (Precision). **Proof Sketch:** *For a program, parallel construction produces the same memory SSA as the sequential version, because (1) the dependences between functions are decoupled by full parameterisation using uniquely identified regions, whose alias sets are disjoint, and (2) all functions assigned to threads are handled using the same memory SSA construction algorithm.*

3. Evaluation

The objective is to show that our parallel memory SSA construction algorithm is significantly faster than the sequential one in analysing large-scale real-world applications with millions of lines of code.

3.1. Implementation

We have fully implemented ParSSA in LLVM-4.0.0. The source files of each benchmark are compiled into bit-code files using clang and then merged together using LLVM Gold Plugin (llv) at link time stage to produce a whole program bitcode file. The compiler flag mem2reg is applied to promote memory into registers.

We use flow-insensitive and field-sensitive Andersen's analysis (Sui and Xue, 2016b) as pre-analysis to generate memory regions. The call graph of a program is constructed on-the-fly during points-to resolution. Our handling of field-sensitivity is ANSI-compliant (ISO90, 1990). The fields of an struct object are distinguish by their unique indices. ParSSA adopts a field-index-based approach to field-sensitivity similar as Pearce et al. (2007).

```
   Function ConstructMemSSA(f)
   begin
1      CreateMuChi(f)
2      InsertPhi(f)
3      SSARename(f)
   Function CreateMuChi(f)
   begin
4      foreach bb ∈ f do
5          foreach (ℓ : 𝕌; 𝔻) ∈ bb do
6              foreach R ∈ ϒ ∧ (R ∩ 𝕌 ≠ ∅) do
7                  add μ(R) before ℓ
8              foreach R ∈ ϒ ∧ (R ∩ 𝔻 ≠ ∅) do
9                  add R = χ(R) after ℓ
   Function InsertPhi(f)
   begin
10     let DF(bb) be the dominant frontiers of bb.
11     To reduce the number of inserted φ, we compute GlobalNames which denotes the
       set of regions that live across multiple basic blocks (Blocks(R)). A phi-function
       without user is pruned following Section 9.3.3 in Cooper and Torczon (2011).
12     foreach R ∈ GlobalNames do
13         W ← Blocks(R)
14         while W ≠ ∅ do
15             bb ← W.pop()
16             foreach (bb′ ∈ DF(bb)) && (bb′ ≠ bb) do
17                 insert φ-function for R in bb' ;
18                 W ← W ∪ {bb′};
   Function SSARename(f)
   begin
19     let bb_entry be the basic block at the entry of function f
20     Initialise the stacks (push version 0 of each memory region into its stack)
21     RenameBB(bb_entry);
   Function RenameBB(bb)
   begin
22     foreach (R = φ(...)) ∈ bb do
23         rename R as NewSSAName(R);
24     foreach (μ(R)) ∈ bb or (R′ = χ(R)) ∈ bb following the execution order in bb do
25         rewrite R as top(stack[R]);
26         rewrite R′ as NewSSAName(R);
27     foreach successor in the CFG do
28         fill in φ-function parameters;
29     foreach successor s in the dominator tree do
30         RenameBB(s)
31     foreach (R = χ(..))and(R = φ(..)) ∈ bb do
32         Pop(stack[R])
   Function NewSSAName(R)
   begin
33     i ← counter[R];
34     counter[R] ← counter[R] + 1;
35     push R_i onto stack[R];
36     return R_i
```

Algorithm 3. Memory SSA construction based for a parameterised function with μ/χ annotations Cooper and Torczon (2011)

```
   Function Main(ThreadCount)
   begin
1      task_group g(ThreadCount);
2      foreach task ∈ g do
3          task.run(ParallelConstruct); // parallel execution
   Function ParallelConstruct()
   begin
4      ConcurrentQueue is initialised with all the functions of a program
5      while ConcurrentQueue ≠ ∅ do
6          f ← ConcurrentQueue.pop()
7          ConstructMemSSA(f)
```

Algorithm 4. Parallel memory SSA construction

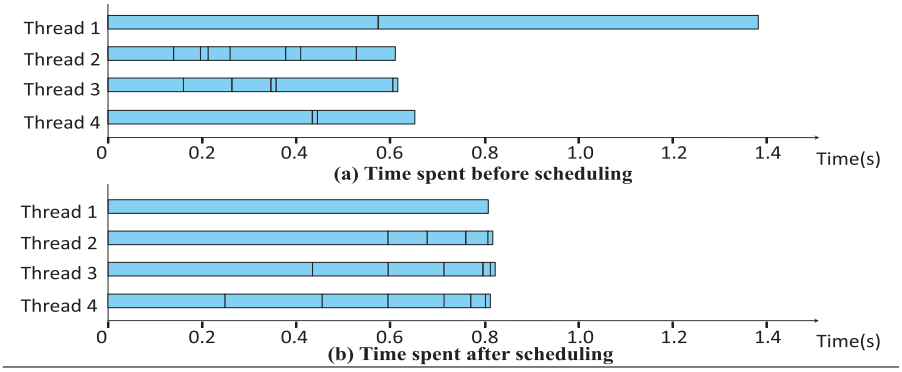


Fig. 5. A scenario adapted from `a2ps` with four threads launched for memory SSA construction, where each rectangle represents a task and the length of the rectangle denotes the size of the task. An uneven workload assignment is shown in (a), while an optimised scheduling is illustrated in (b).

Table 1
Program characteristics (#Call and #Fun denote the numbers of callsites and functions respectively).

Program	KLOC	#AddrOf	#Load	#Store	#Copy	#Call	#Fun	#Pointer
make	40.4	2297	2344	1244	13246	2243	765	36707
a2ps	64.6	3746	6713	1965	29909	3760	917	116129
bison	113.3	3024	8363	3453	42411	4538	566	90049
tar	132.0	4055	4095	2522	24440	2962	815	85727
bash	155.9	8069	10456	4500	43868	10831	3422	191413
sendmail	259.9	7044	6805	3310	46269	14957	1513	256074
python	431.9	16235	34068	14997	119693	33379	9585	412764
vim	330.1	13450	27764	15849	129336	22072	4794	466493
emacs	413.1	15010	29712	20932	160475	16180	1058	754746
gdb	1818.1	54863	66580	45563	406052	104224	21597	191413
dealII	199.0	59305	72373	59305	61272	109085	19477	808614
omnetpp	48.0	14346	13797	10923	13363	19686	2874	151024
povray	155.0	17066	33755	10441	4795	17872	2123	269772
soplex	41.0	7978	10338	4142	5965	9519	1597	102025
xalan	553.0	132258	145409	94483	86073	131196	29973	1080028
Total	4755.3	358746	472572	293629	1187167	502504	101076	5012978

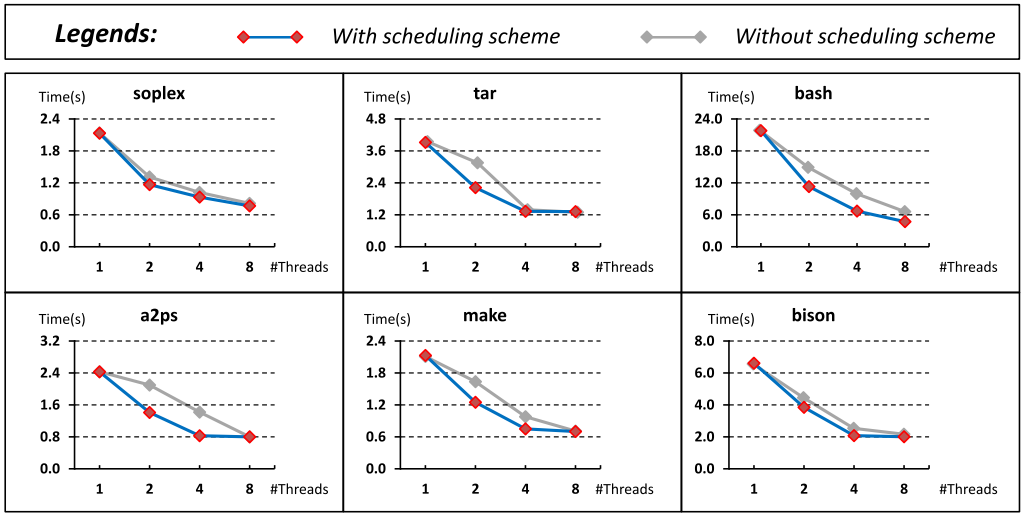
Table 2
Memory SSA statistics. LoadMu, RetMu and CSMu denote μ functions at loads, function exits and callsites, respectively. StoreChi, EntryChi and CSChi denote χ functions annotated at stores, function entries and callsites, respectively.

Program	Number of μ			Number of χ			#Phi
	#LoadMu	#RetMu	#CSMu	#StoreChi	#EntryChi	#CSChi	
make	6756	1729	11467	1455	1809	6732	5542
a2ps	10491	3124	8072	1941	3133	4336	3616
bison	15265	4553	29220	4526	4570	17712	19089
tar	6378	4557	27792	2381	4644	13446	13267
bash	21761	12098	63883	5174	12172	44453	32590
sendmail	16864	4206	115449	5146	4266	60950	34529
python	70621	20327	153771	17066	20348	85892	58992
vim	48622	16784	175470	18368	16792	142099	109803
emacs	43206	21943	200895	18587	22077	149505	140367
gdb	115828	48354	421505	41901	48764	325962	148924
dealII	74574	94255	220773	56700	94606	79913	137856
omnetpp	16369	12901	68962	10516	13067	46757	33700
povray	39812	11789	44484	10311	11799	27748	18951
soplex	13759	5605	11731	4172	5605	3622	4453
xalan	138912	131668	232589	88029	133530	101087	134386
Total	639218	393893	1786063	286273	397182	960709	755698

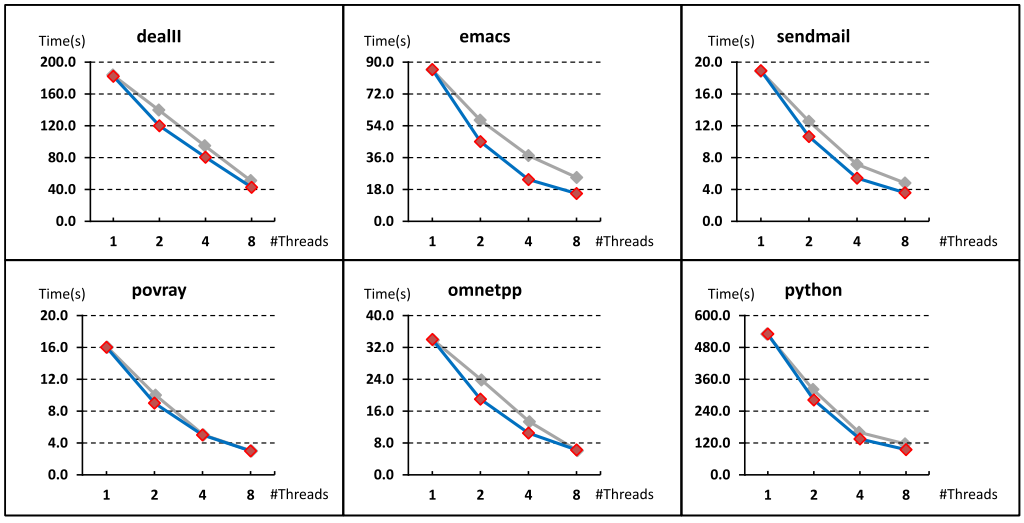
For a struct allocation $p{=}\&o$, a field-insensitive object o is created to represent the entire struct object. A field object o_{fld} is derived from o when analyzing a field access $q{=}\&p{\rightarrow}fld$, where fld is a constant. Thus, different fields (including index 0) are modeled using distinct (sub) objects. Two pointer dereferences are aliased if one refers to o and another one refers to one of its fields e.g., o_{fld} since it is the sub component of o. However, dereferences refer to different fields of o are distinguished and not aliased.

For a pointer arithmetic $q{=}p+i$, if p points to a struct object, we conservatively treat that q can point any field of this struct object. This is based on the ANSI-compliant assumption that i is not across the boundary of the object. A pointer arithmetic used for accessing an aggregate object out of the boundary may cause unsoundness. Arrays are treated monolithically, i.e., accessing any element of an array is treated as accessing the entire array object.

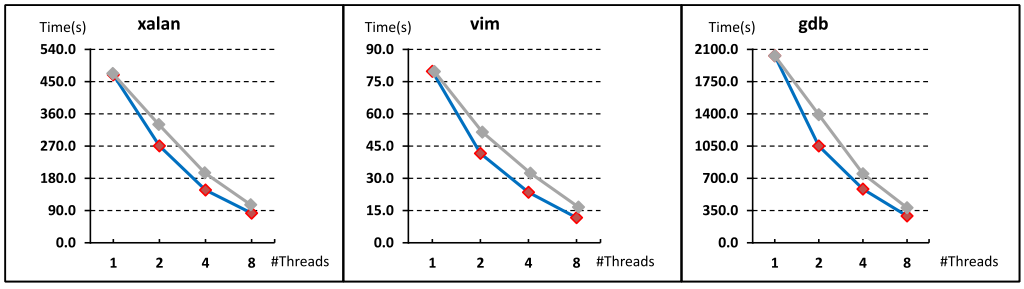
The parallelisation scheme can be summarised as a thread pool pattern (Pool, 2018). We use Intel's Threading Building Blocks li-



(a) Benchmarks with maximum speedup less than 4x



(b) Benchmarks with maximum speedup between 4x and 5.6x



(c) Benchmarks with maximum speedup more than 5.6x

Fig. 6. Memory SSA construction time under different thread configurations.

brary (TBB) to allocate multiple threads for parallel construction of memory SSA as in Algorithm 3. The *concurrent_queue* data structure is used to store all parameterised functions. We use *task_group* to allocate parallel threads for constructing per-function memory SSA from a *concurrent_queue* data structure.

3.2. Methodolgy

We evaluate ParSSA using 15 real-world applications including 10 large open-source C programs and 5 large C++ programs as listed in Table 1: `make` (a build automation tool), `a2ps` (a postScript filter), `bison` (a parser), `tar` (tar archiving), `bash` (a

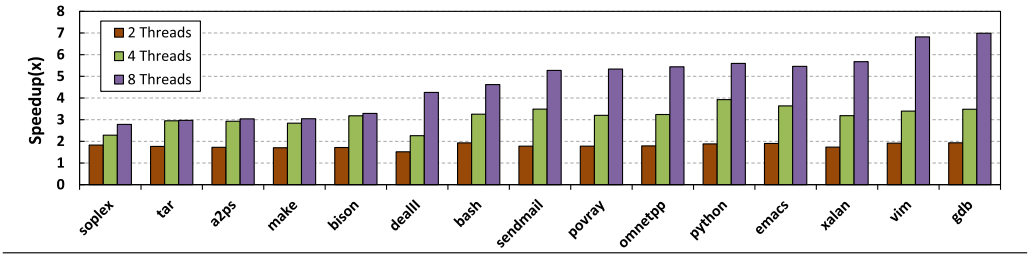


Fig. 7. Speedups achieved with two, four and eight threads.

unix shell and command language), `sendmail` (an email server and client), `python` (a scripting language), `vim` (a text editor), `emacs` (a text editor), `gdb` (linux debugger), `dealII` (finite element analysis), `omnetpp` (discrete event simulation), `povray` (image ray-tracing), `soplex` (linear programming) and `xalan` (XML process).

There are altogether over 4.7 million lines with the largest one, `gdb`, comprising over 1.8 million LOC. Experiments are conducted on a computer with 3.7G Hz Intel Xeon 8-core CPU and 16 GB memory, running Ubuntu Linux (kernel version 3.11.0). For each benchmark, we evaluate the performance advantages of our parallel implementation with two, four and eight threads (TBB tasks) enabled over the sequential one for constructing memory SSA of the same program in Table 1.

A scheduling strategy is implemented to optimise parallel task allocation. In order to avoid idle threads and workload imbalance, the goal of our strategy is to evenly assign workload to each thread, so that a better performance can be achieved. However, optimal scheduling as a classic *partition problem* (Gent and Walsh, 1998) is NP-complete. We have implemented a greedy algorithm introduced in Korf (2009) to produce results close to optimal in polynomial time ($O(n\log n)$). Fig. 5 shows an example adapted from `a2ps`, which illustrates how the optimised scheduling can reduce the execution time. In ParSSA, each task (workload) corresponds to a function in the program. The size of each workload for a function f is estimated by the total number of the annotated μ, χ and ϕ functions in f.

3.3. Results and analysis

Table 2 shows the numbers of annotated μ, χ and ϕ functions for memory regions in each program. For the 15 programs evaluated, 5,219,036 annotations are added in total, with 1,151,238 added in the largest program `gdb`.

Fig. 6 gives memory SSA construction times under three different configurations (with thread counts being 1, 2, 4 and 8). For each program, we run every configuration five times and report the average time.

The blue lines in Fig. 6 represent the SSA construction time (seconds) using the optimised scheduling strategy (described in Section 3.2). The average speedups gained with two, four and eight threads are 1.79X, 3.15X and 4.71X respectively. The grey lines represent the construction time without the scheduling scheme. The performance results are worse when disabling the scheduling strategy, resulting in the speedups of 1.60x, 3.01x, 4.57x under the three configurations. This is caused by the imbalanced workloads for different parallel tasks, especially for some programs, e.g., `emacs`, `a2ps`, `vim` and `gdb` whose function sizes vary significantly. The function sizes of most C++ programs (e.g., `soplex`, `dealII`, `povary` and `xalan`) tends to be more balanced due to the object-oriented design patterns.

The total construction time of all the benchmarks by a sequential algorithm is significantly reduced from 3486.16 to 561.34 seconds using 8 threads under the scheduling strategy. The average speedup for eight threads is 4.7×. The maximum speedup observed is 6.9× (`gdb`). These results are promising, showing that our approach has the potential to be deployed in optimising compilers.

For the four small-size programs, `a2ps`, `bash`, `bison`, `make`, `tar` and `soplex`, the maximum speedups achieved are under 4× (even with 8 threads). For the medium-size programs, `dealII`, `emacs`, `omnetpp`, `povray`, `python` and `sendmail`, which have a relatively large number of pointers and annotations, greater speedups are observed, ranging from 4× to 5.6×, as shown in Fig. 6(b). For the most complex benchmarks, `gdb`, `vim` and `xalan`, all their speedups above 5.6× with 8 threads. In particular, the analysis time for the largest benchmark `gdb` has been cut from 2030.51 seconds to 290.53 seconds.

Fig. 7 compares further the speedups achieved under three different thread configurations. Compared to sequential execution, ParSSA with the scheduling strategy has achieved noticeable speedups for all the benchmarks evaluated, with the best reaching 6.9X (in `gdb`). This demonstrates that ParSSA is effective in accelerating memory SSA construction for large programs.

In general, better speedups are obtained when more threads are used. On average, the speedups gained with 4 threads are 1.74× higher than the speedups gained with 2 threads, while the speedups gained with 8 threads are 1.47× higher than the speedups gained with 4 threads. However, it worth noting that for small applications (e.g., `a2ps`, `make` and `soplex`), using 8 threads does not guarantee a better performance than using 4 threads. There are two reasons behind this phenomenon. First, more threads lead to higher synchronisation overheads in accessing the shared data, offsetting the speedups gained from parallelism. Second, for small programs, the overhead of initiating threads is not negligible. In addition, some programs have better speedups than others. The reason is that different programs have different inherent complexities in terms of memory SSA construction, resulting in different synchronisation overheads.

4. Related Work

Static Single Assignment (SSA)

SSA form is the mainstream representation in modern optimising compilers and program analysis tools. Memory SSA advances scalar-based SSA by providing a sparse data-flow representation for both top-level pointers and address-taken variables. Intraprocedural memory SSA forms (Chow et al., 1996; Novillo and Canada, 2007), which approximates conservatively the dependences across the functions is cheaper to compute than their interprocedural counterparts. Recently, the idea of staged analysis (Hardekopf and Lin, 2011; Sui and Xue, 2016a) provides an effective way for using pre-computed points-to information to bootstrap an interprocedural memory SSA. However, the algorithm is still costly for large programs with millions of lines of code.

Parallel Program Analysis

Méndez-Lojo et al. (2010) introduce a parallel implementation of Andersen's pointer analysis for C programs based on graph rewriting. Their parallel analysis is context- and flow-insensitive, achieving a speedup of up to 3× on 8 CPU cores. Recently, the whole-program sparse flow-sensitive pointer analysis (Hardekopf and Lin, 2011) is parallelised on multi-core CPUs (Nagaraj and Govindarajan, 2013) and GPUs (Nasre, 2013). The speedups are up to 2.6× on 8 CPU cores. In their report, Singer and Ward describe a parallel scalar SSA form for Java programs by considering top-level pointers only. To the best of our knowledge, this paper proposes the first approach to parallelising interprocedural memory SSA construction that achieves an average speedup of 4.7× (up to 6.9×) on 8 CPU cores.

5. Conclusion

This paper presented ParSSA, the first parallel memory SSA construction approach by partitioning the whole-program memory objects into uniquely identified memory regions to fully parameterise indirect memory accesses in a function. Thus, the memory dependences across functions are decoupled for parallelising memory SSA construction. Our results show that ParSSA can achieve an average speedup of 4.7× on an 8-core machine, making it deployable in optimising compilers and program analysis tools. There few possible future directions.

There are few interesting directions along this work. One possible future work is to extend ParSSA to support fine-grained parallelism than function level (e.g., basic block and code region level) for constructing memory SSA form. For example, applying region-based analysis (Ye et al., 2014) to parameterise selected program parts for parallel SSA construction. Another interesting direction is to apply the proposed parameterised approach to support parallelising precise pointer analysis (e.g., whole-program flow-sensitive analysis (Sui et al., 2016) and/or demand-driven analysis (Sui and Xue, 2016a)).

References

Andersen, L.O., 1994. Program analysis and specialization for the C programming language. DIKU, PhD Thesis University of Copenhagen.

Cai, H., Santelices, R., Jiang, S., 2016. Prioritizing change-impact analysis via semantic program-dependence quantification. IEEE Trans. Reliab. 65 (3), 1114–1132.

Choi, J.-D., Cytron, R., Ferrante, J., 1994. On the efficient engineering of ambitious program analysis. TSE '94 20 (2), 105–114.

Chow, F., Chan, S., Liu, S., Lo, R., Streich, M., 1996. Effective representation of aliases and indirect memory operations in SSA form. In: CC '96, pp. 253–267.

Cooper, K., Torczon, L., 2011. Engineering a Compiler. Elsevier.

Cytron, R., Ferrante, J., Rosen, B., Wegman, M., Zadeck, F., 1991. Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13 (4), 451–490.

Gent, I.P., Walsh, T., 1998. Analysis of heuristics for number partitioning. Comput. Intell. 14 (3), 430–451.

Guo, S., Kusano, M., Wang, C., 2016. Conc-ise: Incremental symbolic execution of concurrent software. In: ASE '16, pp. 531–542.

Hardekopf, B., Lin, C., 2011. Flow-sensitive pointer analysis for millions of lines of code. In: CGO '11, pp. 289–298.

ISO90, 1990. ISO/IEC. international standard ISO/IEC 9899, programming languages - C.

Korf, R.E., 2009. Multi-way number partitioning.. In: IJCAI '09, pp. 538–543.

Kotzmann, T., Wimmer, C., Mössenböck, H., Rodriguez, T., Russell, K., Cox, D., 2008. Design of the Java hotspot™ client compiler for Java 6. TACO '08 5 (1), 7.

Landi, W., 1992. Undecidability of static analysis. LOPLAS '92 1 (4), 323–337.

Landi, W., Ryder, B.G., 1992. A safe approximate algorithm for interprocedural aliasing. PLDI'92 27 (7), 235–248.

Lattner, C., Adve, V., 2014. LLVM: a compilation framework for lifelong program analysis & transformation. CGO '04 75–86.

Lhoták, O., Chung, K.-C.A., 2011. Points-to analysis with efficient strong updates. In: POPL '11, pp. 3–16.

Livshits, V.B., Lam, M.S., 2003. Tracking pointers with path and context sensitivity for bug detection in C programs. In: FSE '03, pp. 317–326.

Méndez-Lojo, M., Mathew, A., Pingali, K., 2010. Parallel inclusion-based points-to analysis. In: OOPSLA'10, 45, pp. 428–443.

Nagaraj, V., Govindarajan, R., 2013. Parallel flow-sensitive pointer analysis by graph-rewriting. In: PACT '13, pp. 19–28.

Nasre, R., 2013. Time-and space-efficient flow-sensitive points-to analysis. ACM Trans. Archit. Code Optim. (TACO) 10 (4), 39.

Novillo, D., Canada, R.H., 2007. Memory SSA-a unified approach for sparsely representing memory operations. In: Proceedings of the GCC Developers' Summit.

Pearce, D., Kelly, P., Hankin, C., 2007. Efficient field-sensitive pointer analysis of C. ACM Trans. Program. Lang. Syst. 30 (1).

Pool, T., 2018. https://en.wikipedia.org/wiki/thread_pool. Wiki.

Rosen, B.K., Wegman, M.N., Zadeck, F.K., 1988. Global value numbers and redundant computations. POPL '98 12–27.

Singer, J., Ward, M.,. Parallelizing the construction of static single assignment form. http://www.dcs.gla.ac.uk/~jsinger/pdfs/ssaparconstr.pdf.

Sui, Y., Di, P., Xue, J., 2016. Sparse flow-sensitive pointer analysis for multithreaded C programs. in CGO '16.

Sui, Y., Xue, J., 2016. On-demand strong update analysis via value-flow refinement. FSE '16.

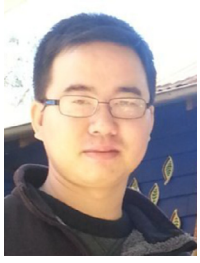
Sui, Y., Xue, J., 2016. SVF: Interprocedural static value-flow analysis in LLVM. CC '16.

Sui, Y., Ye, D., Xue, J., 2012. Static memory leak detection using full-sparse value-flow analysis. In: ISSTA '12, pp. 254–264.

The llvm gold plugin, https://llvm.org/docs/goldplugin.html.

Ye, D., Sui, Y., Xue, J., 2014. Accelerating dynamic detection of uses of undefined variables with static value-flow analysis. In: CGO '14, pp. 154–164.

Ye, S., Sui, Y., Xue, J., 2014. Region-based selective flow-sensitive pointer analysis. In: SAS '14. Springer, pp. 319–336.

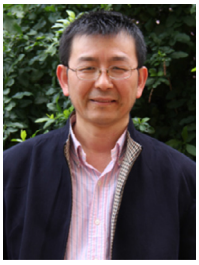


Yulei Sui is a Lecturer (Assistant Professor) and an ARC DECRA at Faculty of Engineering and Information Technology, University of Technology Sydney (UTS). He obtained his Ph.D. from University of New South Wales (UNSW), where he also holds an Adjunct Lecturer position. He is broadly interested in the research field of software engineering and programming languages, particularly interested in static and dynamic program analysis for software bug detection and compiler optimizations. He worked as a software engineer in Program Analysis Group for Memory Safe C project in Oracle Lab Australia. He was an Australian IPRS scholarship holder, a keynote speaker at EuroLLVM and a Best Paper Award winner at CGO, and has been awarded an Australian Discovery Early Career Researcher Award (DECRA) 2017-2019.

Hua Yan is currently a research associate of School of Computor Science and Engineering at University of New South Wales, Sydney, Australia. He obtained his Ph.D. degree from University of New South Wales, Australia, in January 2018. Hua Yan's research interests include software engineering, software security, program analysis, testing and formal verification.

Zheng Zheng is currently an associate professor and the Vice Dean of Department of Automatic Control, School of Automation Science and Electrical Engineering, Beihang University. He received his Ph.D. degree in Computer Software and Theory from Institute of Computing Technology, Chinese Academy of Sciences. He is an Area Editor of International Journal of Computational Intelligence Systems. He has served as an investigator on Beijing Youth Talent Project of China, and was honoured by The Dean Scholarship of Chinese Academy of Science.

Yupeng Zhang received his Ph.D. degree in computer science from Northwestern Polytechnical University, China, in 2009. He is currently working as an Assistant Professor at University of Houston. Dr. Zhang has worked at Boise State University and Dakota State University (U.S.), University of Melbourne (Australia), Imperial College London (U.K.) and Northwestern Polytechnical University (China) as a Cybersecurity Expert for more than 15 years. He has published 49 papers in peer-reviewed journals and conference and served as a program committee member for many conferences in his area.



Jingling Xue received his B.Sc. and M.Sc. degrees in Computer Science and Engineering from Tsinghua University in 1984 and 1987, respectively, and his Ph.D. degree in Computer Science and Engineering from Edinburgh University in 1992. He is currently a Profein the School of Computer Science and Engineering, University of New South Wales, Australia, where he heads the Programming Languages and Compilers Group. His main research interest has been programming languages and compilers for about 20 years. He is currently supervising a group of postdocs and Ph.D. students on a number of topics including programming and compiler techniques for multicore processors and embedded systems, concurrent programming models, and program analysis for detecting bugs and security vulnerabilities. He is presently an Associate Editor of IEEE Transactions on Computers, Software: Practice and Engineering, International Journal of Parallel, Emergent and Distributed Systems, and Journal of Computer Science and Technology. He has served in various capacities on the Program Committees of many conferences in his field.