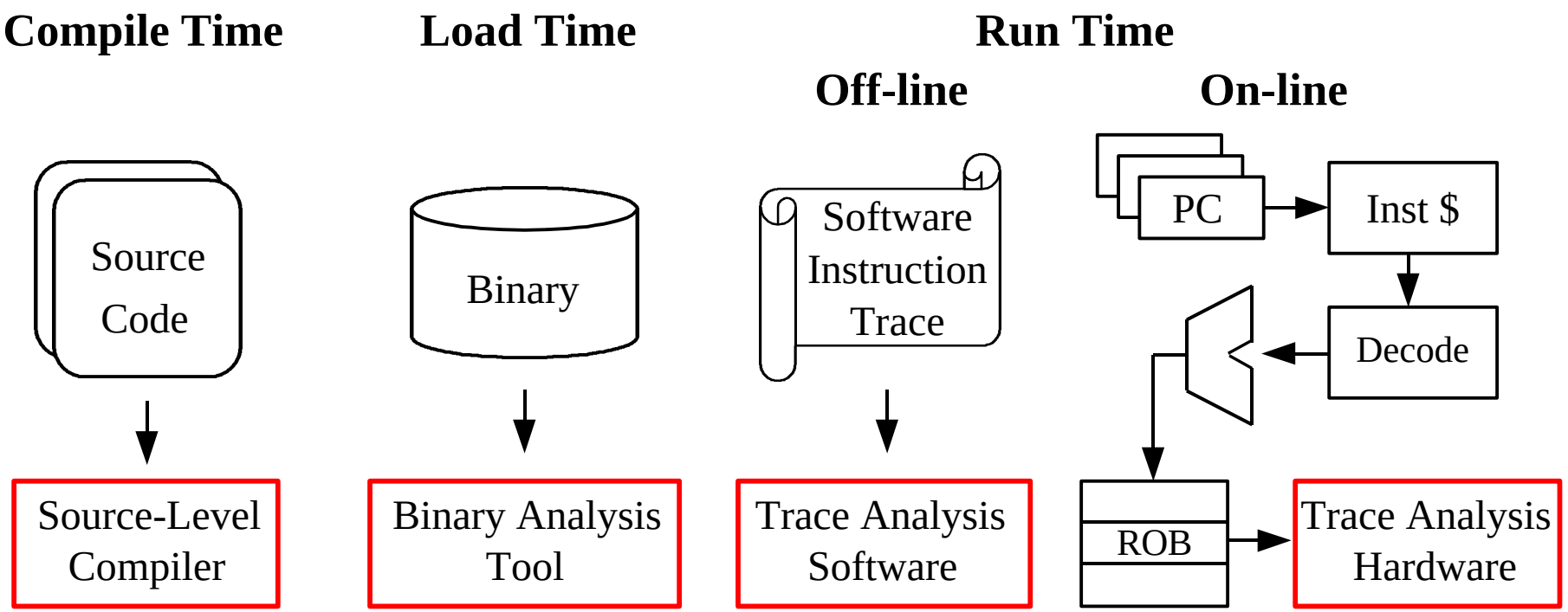
Compiler Support for Speculative Precomputation

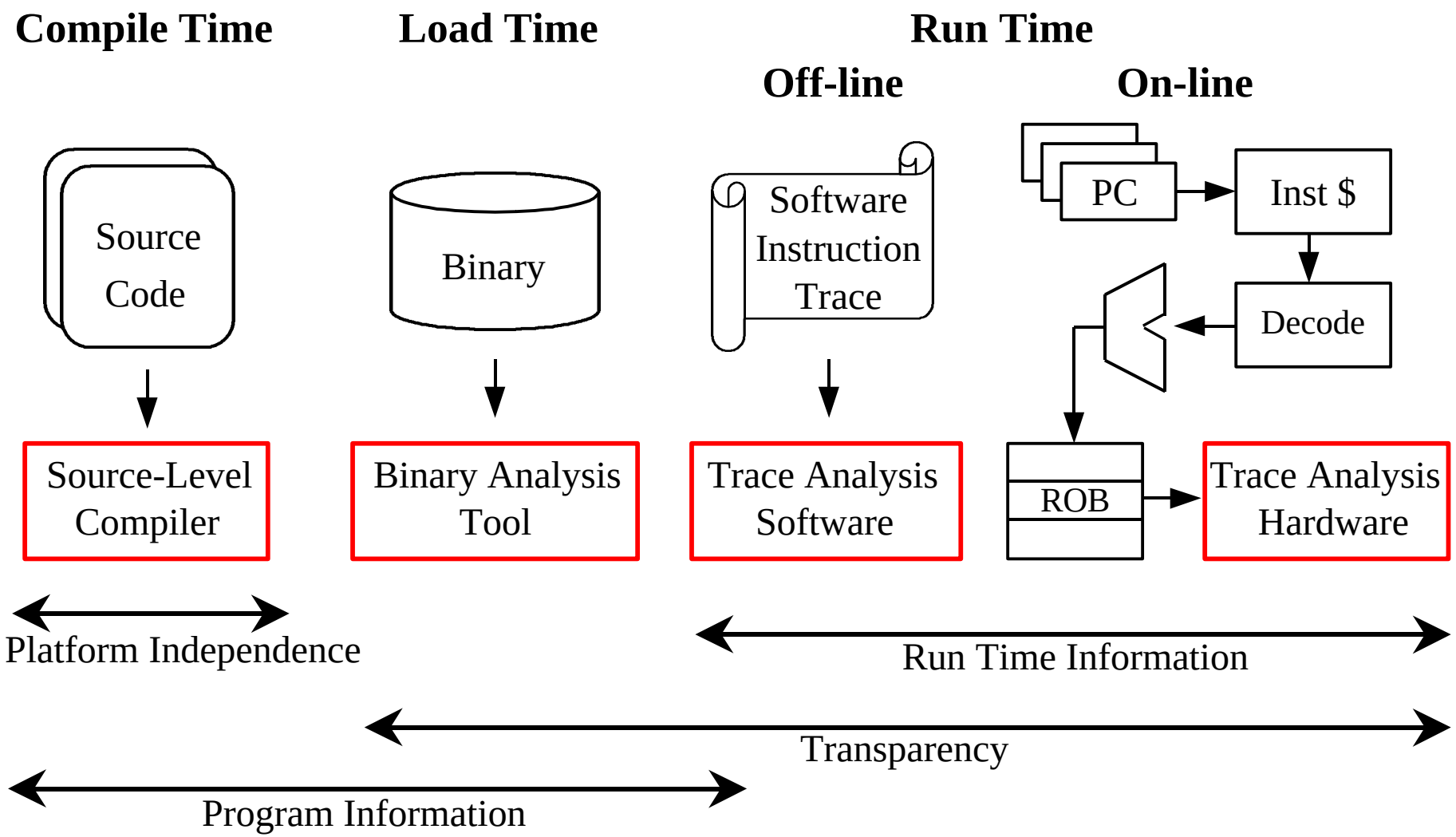
Donald Yeung

Department of Electrical and Computer Engineering
University of Maryland at College Park

Alternatives for P-slice Extraction



Alternatives for P-slice Extraction



Outline

- Compiler-Based Speculative Precomputation
  - Optimizations
  - Implementation
  - Evaluation
- Alternate Compiler Algorithms
  - Simplify Program Slicing
  - Eliminate Profiles
- Software-Controlled Pre-Execution

Compiler-Based Speculative Precomputation

Main Program Code

```
void foo(…) {
   doall(clone_precomp, …);
   for (i = start; i <= end; i++) {
      … = B[A[i]];
   }
   kill();
}
```

Clone

Precomputation Code

```
void clone_precomp(…) {
   for (i = start; i <= end; i++) {
      … = B[A[i]];
   }
}
```

Identification

- Cache Miss Profile

Accuracy

- Code Cloning
- Precomputation Regions

Timeliness

- Program Slicing
- Prefetch Conversion
- Speculative Loop Parallelization

Correctness

- Store Removal
- Exception Handling
- Kill

Program Slicing

- Analogous to backward slicing
- Remove unnecessary code at source level
- Originally for program understanding (Mark Weiser, 1984)

➡ Extract memory-driven slices at source-code level

Unravel

- Developed by Lysle *et al*, NIST, 1995
- Basic analysis

$$S_{m,v} = \begin{cases} S_{n,v} & v \notin defs(n) \\ \{n\} \cup \left(\bigcup_{x \in refs(n)} S_{n,x} \right) & \text{otherwise} \end{cases}$$

- Advanced analysis
  - Array index and structure field analysis
  - Pointer analysis
  - Inter-procedure analysis

Program Slicing for Precomputation

```
void BlueRule(Vtx inserted, Vtx vlist) {
  Vertex tmp,prev,next;
  Hash hash;
  int dist, dist2;

  for (tmp=vlist->next; tmp;
       prev=tmp,tmp=tmp->next) {
    if (tmp==inserted) {
      next = tmp->next;
      prev->next = next;
    } else {
      hash = tmp->edgehash;
      dist2 = tmp->mindist;
      dist = HashLookup(inserted, hash);
      if (dist) {
        if (dist < dist2) {
          tmp->mindist = dist;
          dist2 = dist;
        }
      } else printf("Not found\n");
      if (dist2 < retval.dist) {
        retval.vert = tmp;
        retval.dist = dist2;
      }
    }
  }
}
```

```
void *HashLookup(int key, Hash hash) {
  int j;
  HashEntry ent;

  j = (hash->mapfunc)(key);
  for (ent=hash->tab[j]; ent && ent->key!=key;
       ent=ent->next);
  if (ent) return ent->entry;
  return NULL;
}
```

1. Problematic Memory Reference

Program Slicing for Precomputation

```
void BlueRule(Vtx inserted, Vtx vlist) {
  Vertex tmp,prev,next;
  Hash hash;
  int dist, dist2;

  for (tmp=vlist->next; tmp;
       prev=tmp,tmp=tmp->next) {
    if (tmp==inserted) {
      next = tmp->next;
      prev->next = next;
    } else {
      hash = tmp->edgehash;
      dist2 = tmp->mindist;
      dist = HashLookup(inserted, hash);
      if (dist) {
        if (dist < dist2) {
          tmp->mindist = dist;
          dist2 = dist;
        }
      } else printf("Not found\n");
      if (dist2 < retval.dist) {
        retval.vert = tmp;
        retval.dist = dist2;
      }
    }
  }
}
```

```
void *HashLookup(int key, Hash hash) {
  int j;
  HashEntry ent;

  j = (hash->mapfunc)(key);
  for (ent=hash->tab[j]; ent && ent->key!=key;
       ent=ent->next);
  if (ent) return ent->entry;
  return NULL;
}
```

1. Problematic Memory Reference
2. Side-Effect Store

Program Slicing for Precomputation

```
void BlueRule(Vtx inserted, Vtx vlist) {



  for (tmp=vlist->next; tmp;
       prev=tmp,tmp=tmp->next) {
    if (tmp==inserted) {


    } else {
      hash = tmp->edgehash;

      dist = HashLookup(inserted, hash);









    }
  }
}
```

```
void *HashLookup(int key, Hash hash) {



  j = (hash->mapfunc)(key);
  for (ent=hash->tab[j]; ent && ent->key!=key;
       ent=ent->next);


}
```

1. Problematic Memory Reference
2. Side-Effect Store
3. Perform Slicing

Program Slicing for Precomputation

```
void BlueRule(Vtx inserted, Vtx vlist) {



  for (tmp=vlist->next; tmp;
       prev=tmp,tmp=tmp->next) {
    if (tmp==inserted) {


    } else {
      hash = tmp->edgehash;
      dist2 = tmp->mindist;
      dist = HashLookup(inserted, hash);









    }
  }
}
```

```
void *HashLookup(int key, Hash hash) {



  j = (hash->mapfunc)(key);
  for (ent=hash->tab[j]; ent && ent->key!=key;
       ent=ent->next);


}
```

1. Problematic Memory Reference
2. Side-Effect Store
3. Perform Slicing
4. Merge Slices

Program Slicing for Precomputation

```
void BlueRule(Vtx inserted, Vtx vlist) {



  for (tmp=vlist->next; tmp;
       prev=tmp,tmp=tmp->next) {
    if (tmp==inserted) {


    } else {
      hash = tmp->edgehash;
      dist2 = tmp->mindist;
      asm(  :  : r dist2);
      dist = HashLookup(inserted, hash);









    }
  }
}
```

```
void *HashLookup(int key, Hash hash) {



  j = (hash->mapfunc)(key);
  for (ent=hash->tab[j]; ent && ent->key!=key;
       ent=ent->next);
     asm(  :  : r  ent->key);

}
```

1. Problematic Memory Reference
2. Side-Effect Store
3. Perform Slicing
4. Merge Slices
5. Code Pinning

Prefetch Conversion

```
void BlueRule(Vtx inserted, Vtx vlist) {



  for (tmp=vlist->next; tmp;
       prev=tmp,tmp=tmp->next) {
    if (tmp==inserted) {


    } else {
      hash = tmp->edgehash;
      dist2 = tmp->mindist;
      asm(  .  : : .r  &tmp->mindist);
      dist = HashLookup(inserted, hash);









    }
  }
}
```

```
void *HashLookup(int key, Hash hash) {



  j = (hash->mapfunc)(key);
  for (ent=hash->tab[j]; ent && ent->key!=key;
       ent=ent->next);
     asm(  .  : : .r  &ent->key);

}
```

- Reduce blocking in pre-execution threads
- Enabled by program slicing analysis

Prefetch Conversion Candidate

Precomputation Initiation Schemes

- Three precomputation initiation schemes:

SERIAL

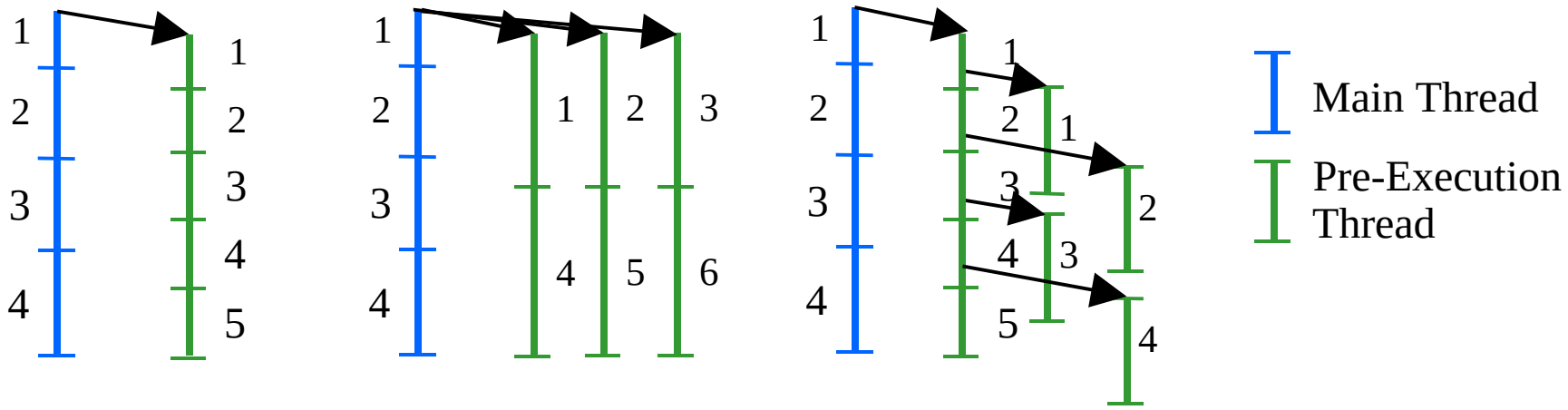
```
for (; ;)
/* Non-Blocking */
```

DOALL

```
for (; ; i += 3)
/* Blocking */
```

DOACROSS

```
for (; ; ptr=ptr ➔next)
/* Blocking */
```



- Speculative Loop Parallelization
  - Only analyze loop induction variables

Selecting Precomputation Regions

- Precomputation regions are loops
- Inner-most loop or Next-outer loop?
  - Inner-most loop: Reduce loop-carried dependence
  - Next-outer loop: Reduce initiation overhead

```
for (i = start; i <= end; i++) {
    for (j = start; j <= end; j++) {
        … = A[i][j];
    }
}
```

Next-outer loop

Inner-most loop

- Precomputation region selection algorithm

Selecting Precomputation Regions

Global Loop-Nest Graph

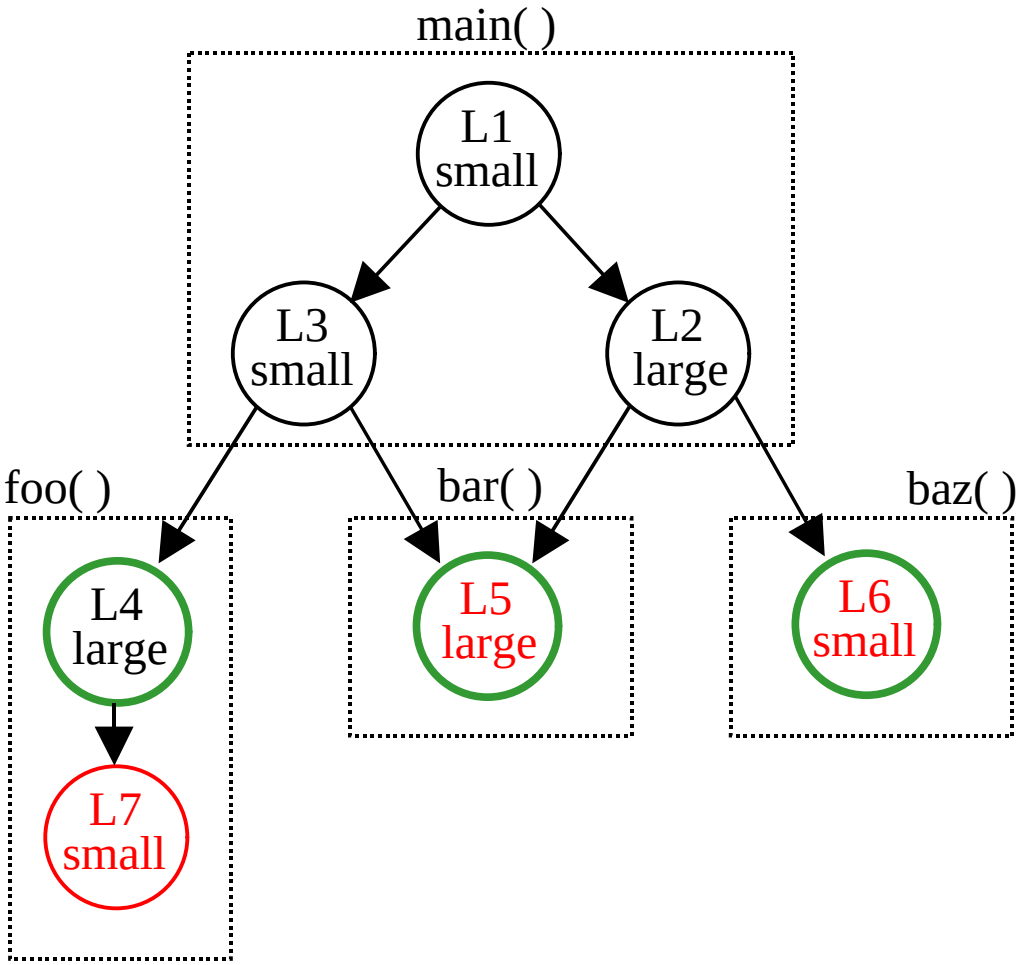
Problematic loads

Loop-trip count (small or large)

Select large inner-most loops

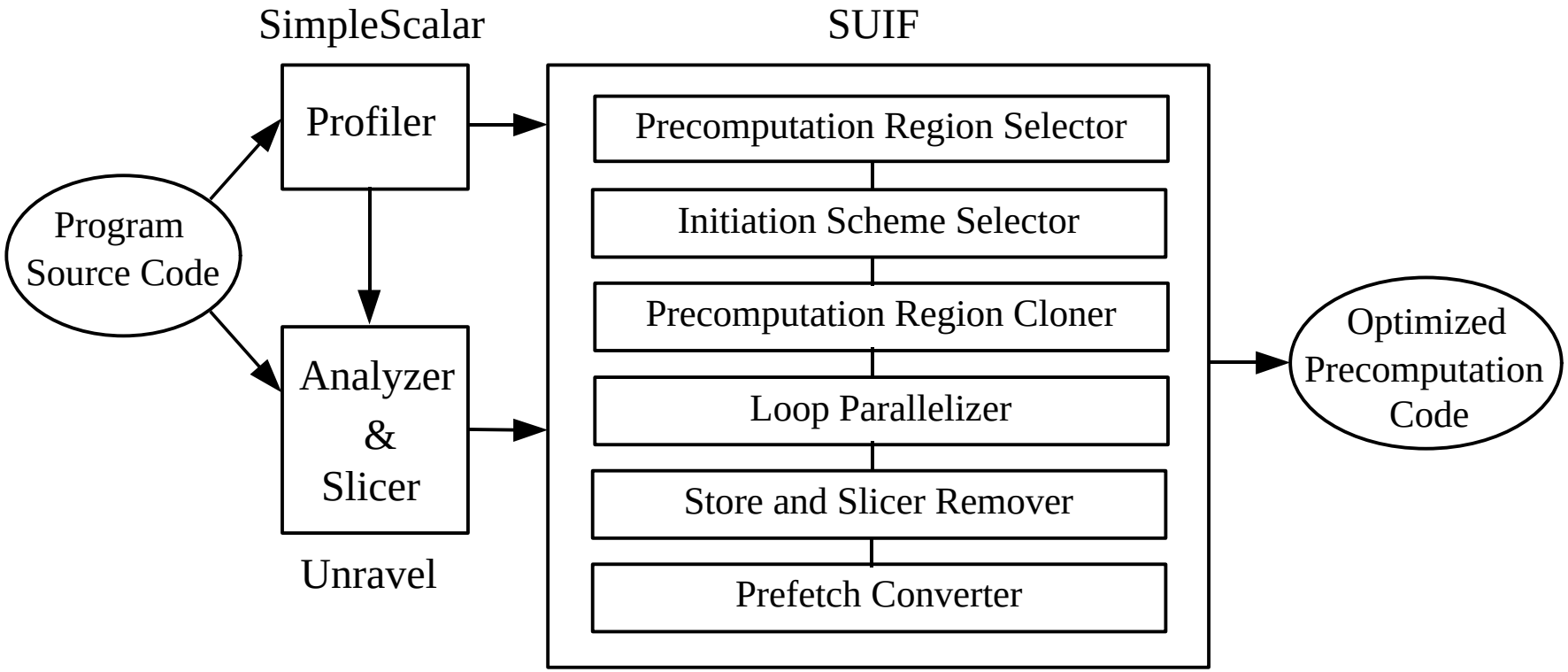
Select next-outer loops without nested inner-most loop

Select remaining inner-most loops



Compiler Prototype

- Prototype compiler algorithms in SUIF
- Leverage SimpleScalar and Unravel



Actual Code Example (EQUAKE)

```
void smvp(int nodes, double ***A, int *Acol, int *Aindex, double **v, double **w) {
  int i;
  int Anext, Alast, col;
  double sum0, sum1, sum2;

  resumeID = loop_37;
  main_counter = 24;
  *my_array = nodes;
  my_array[1] = (int)Aindex;
  my_array[2] = (int)A;
  my_array[3] = (int)v;
  my_array[4] = (int)Acol;
  my_array[5] = (int)w;
  resumeContext(1);
  resumeContext(2);
  resumeContext(3);

  for (i = 0; i < nodes; i++) {
    main_counter = main_counter + 1;
    Anext = Aindex[i];
    Alast = Aindex[i + 1];

    sum0 = A[Anext][0][0]*v[i][0] + A[Anext][0][1]*v[i][1] + A[Anext][0][2]*v[i][2];
    sum1 = A[Anext][1][0]*v[i][0] + A[Anext][1][1]*v[i][1] + A[Anext][1][2]*v[i][2];
    sum2 = A[Anext][2][0]*v[i][0] + A[Anext][2][1]*v[i][1] + A[Anext][2][2]*v[i][2];

    Anext++;
    while (Anext < Alast) {
      col = Acol[Anext];

      sum0 += A[Anext][0][0]*v[col][0] + A[Anext][0][1]*v[col][1] + A[Anext][0][2]*v[col][2];
      sum1 += A[Anext][1][0]*v[col][0] + A[Anext][1][1]*v[col][1] + A[Anext][1][2]*v[col][2];
      sum2 += A[Anext][2][0]*v[col][0] + A[Anext][2][1]*v[col][1] + A[Anext][2][2]*v[col][2];

      w[col][0] += A[Anext][0][0]*v[i][0] + A[Anext][1][0]*v[i][1] + A[Anext][2][0]*v[i][2];
      w[col][1] += A[Anext][0][1]*v[i][0] + A[Anext][1][1]*v[i][1] + A[Anext][2][1]*v[i][2];
      w[col][2] += A[Anext][0][2]*v[i][0] + A[Anext][1][2]*v[i][1] + A[Anext][2][2]*v[i][2];
      Anext++;
    }
    w[i][0] += sum0;
    w[i][1] += sum1;
    w[i][2] += sum2;
  }
  SSMT_EXIT();
}
```

```
extern void loop_37(int my_tid) {
  int i, Anext, Alast, col, *Acol, *Aindex, nodes, local_counter, consumer_counter;
  double sum0, sum1, sum2, **w, ***A, **v;

  w = (double **)my_array[5];
  Acol = (int *)my_array[4];
  v = (double **)my_array[3];
  A = (double ***)my_array[2];
  Aindex = (int *)my_array[1];
  nodes = *my_array;
  local_counter = my_tid - 1;

  for (i = 0 + my_tid - 1; i < nodes; i += 3) {
    do {
      NOREORDER(asm volatile ("lw %0, 0(%1)" : "=r" (consumer_counter) : "r" (&main_counter)));
    } while (local_counter >= consumer_counter);
    local_counter = local_counter + 3;

    Anext = Aindex[i];
    Alast = Aindex[i + 1];
    sum0 = **A[Anext] * *v[i] + (*A[Anext])[1] * v[i][1] + (*A[Anext])[2] * v[i][2];
    sum1 = *A[Anext][1] * *v[i] + A[Anext][1][1] * v[i][1] + A[Anext][1][2] * v[i][2];
    sum2 = *A[Anext][2] * *v[i] + A[Anext][2][1] * v[i][1] + A[Anext][2][2] * v[i][2];

    Anext = Anext + 1;
    if (Anext < Alast) {
      do {
        double *suif_tmp, *suif_tmp1, *suif_tmp3;
        col = Acol[Anext];

        sum0 = sum0 + **A[Anext] * *v[col] + (*A[Anext])[1] * v[col][1] + (*A[Anext])[2] * v[col][2];
        sum1 = sum1 + *A[Anext][1] * *v[col] + A[Anext][1][1] * v[col][1] + A[Anext][1][2] * v[col][2];
        sum2 = sum2 + *A[Anext][2] * *v[col] + A[Anext][2][1] * v[col][1] + A[Anext][2][2] * v[col][2];

        suif_tmp = w[col]; prefetch(&v[i][2]); prefetch(A[Anext][2]); prefetch(&v[i][1]);
        prefetch(A[Anext][1]); prefetch(v[i]); prefetch(*A[Anext]); prefetch(suif_tmp);
        suif_tmp1 = &w[col][1]; prefetch(&v[i][2]); prefetch(&A[Anext][2][1]); prefetch(&v[i][1]);
        prefetch(&A[Anext][1][1]); prefetch(v[i]); prefetch(&(*A[Anext])[1]); prefetch(suif_tmp1);
        suif_tmp3 = &w[col][2]; prefetch(&v[i][2]); prefetch(&A[Anext][2][2]); prefetch(&v[i][1]);
        prefetch(&A[Anext][1][2]); prefetch(v[i]); prefetch(&(*A[Anext])[2]); prefetch(suif_tmp3);
        Anext = Anext + 1;
      } while (Anext < Alast);
    }
    prefetch(w[i]); prefetch(&w[i][1]); prefetch(&w[i][2]);
  }
}
```

Experimental Methodology

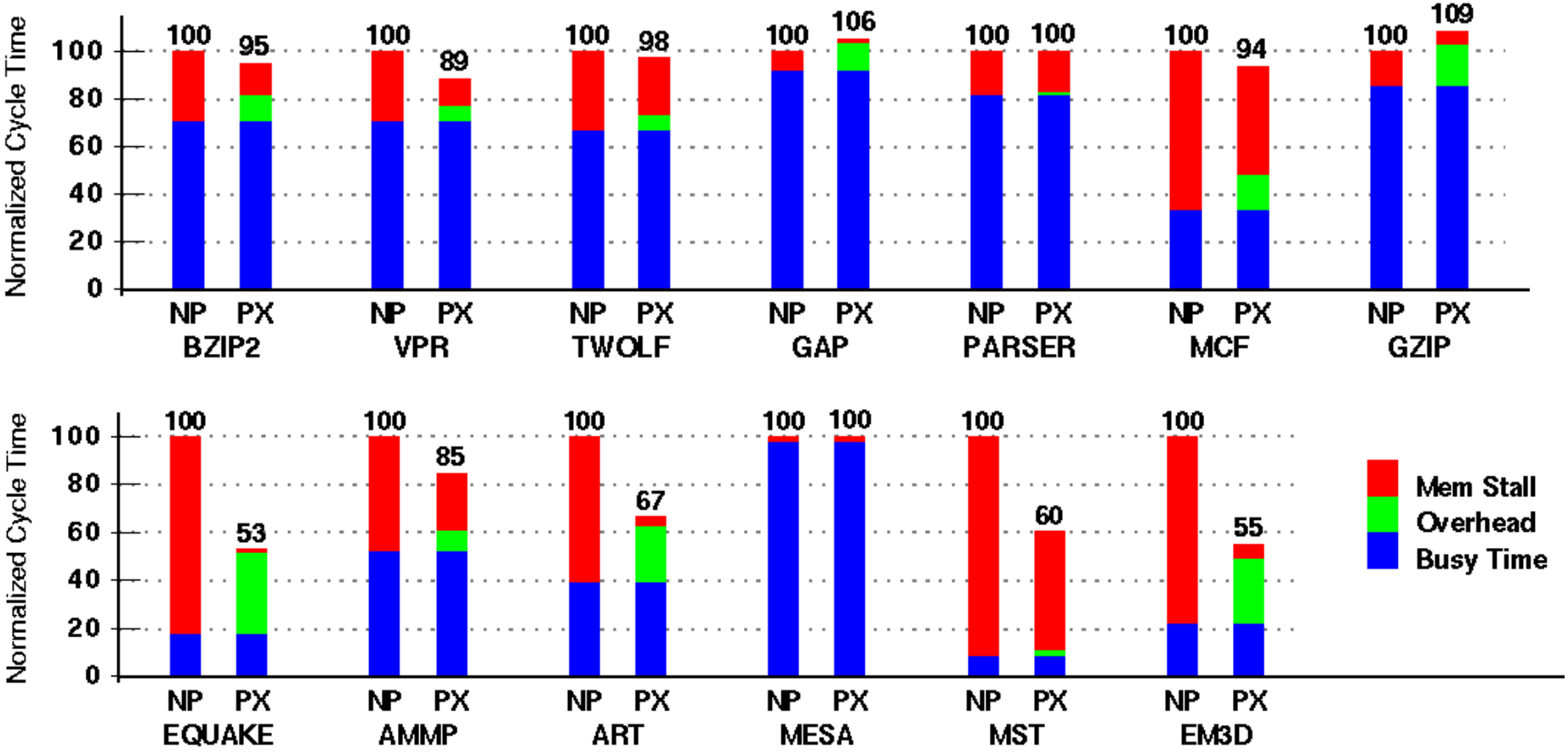
- SMT simulator derived from SimpleScalar

# Contexts	4	Issue Width	8-way
IFQ Size	32	LSQ Size	64
ROB Size	128	Int/FP FU	8/4
BR Predictor	2K-Gshare	BTB Size	2K
L1 Cache Size	32KB	L2 Cache Size	1MB
L1/L2 Latency	1/10 cycles	Memory Latency	122 cycles

- Benchmark Suite (13 apps)

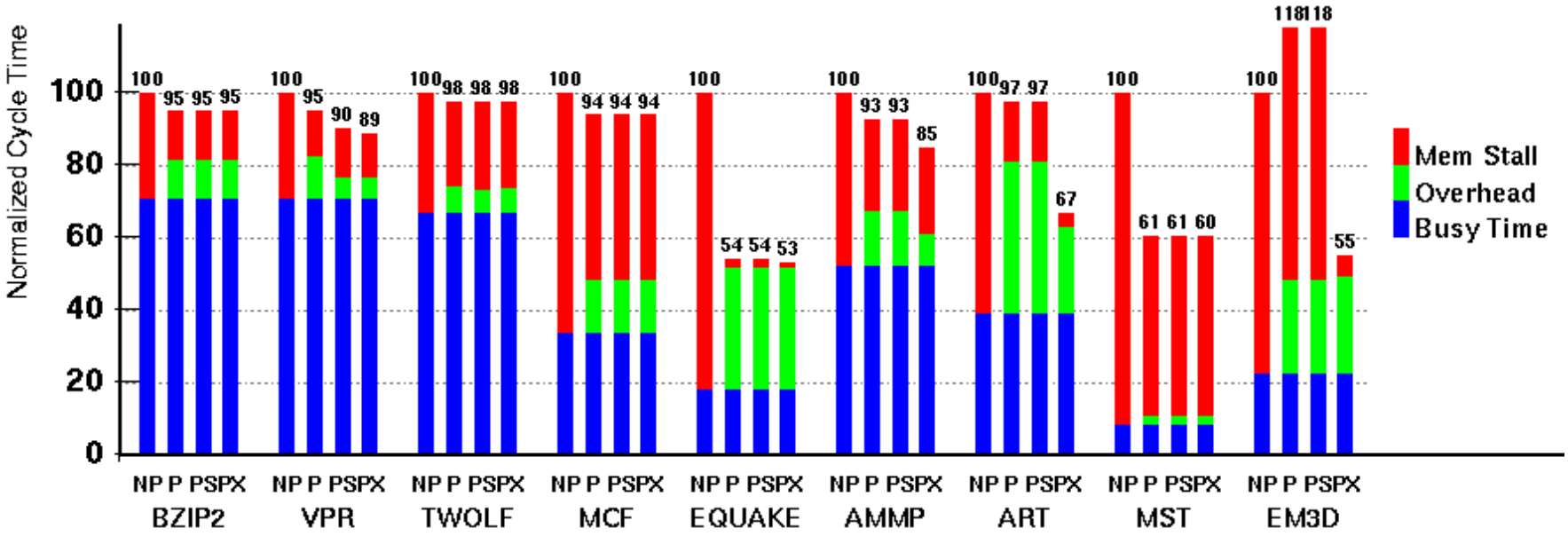
SPECint 2000	BZIP2, VPR, TWOLF, GAP, PARSER, MCF, GZIP
SPECfp 2000	EQUAKE, AMMP, ART, MESA
Olden	MST, EM3D

Baseline Result



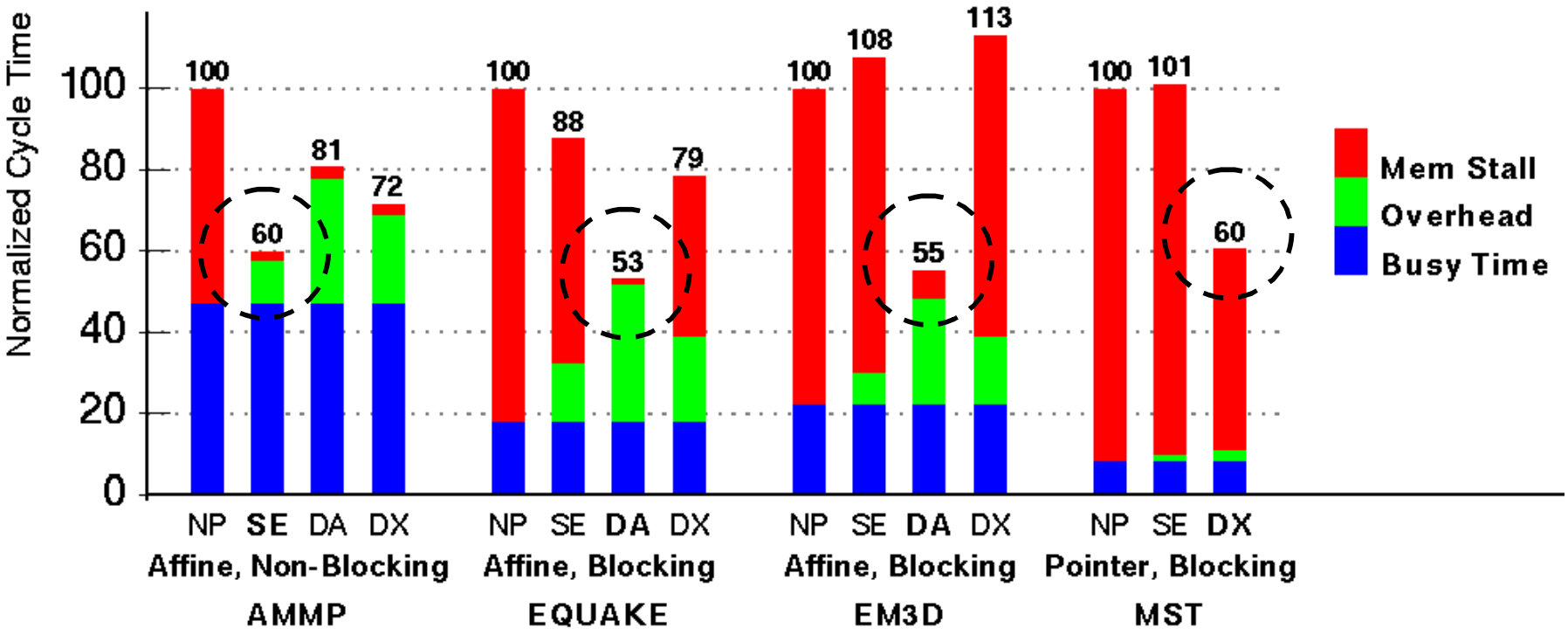
- Execution time reduced by 23% for 9 of 13 applications
- 17% overall speedup

Evaluation of Optimizations



- Speculative loop parallelization is important for 8 applications
- Program slicing is important for 1 application
- Prefetch conversion is important for 4 applications

Precomputation Initiation Schemes

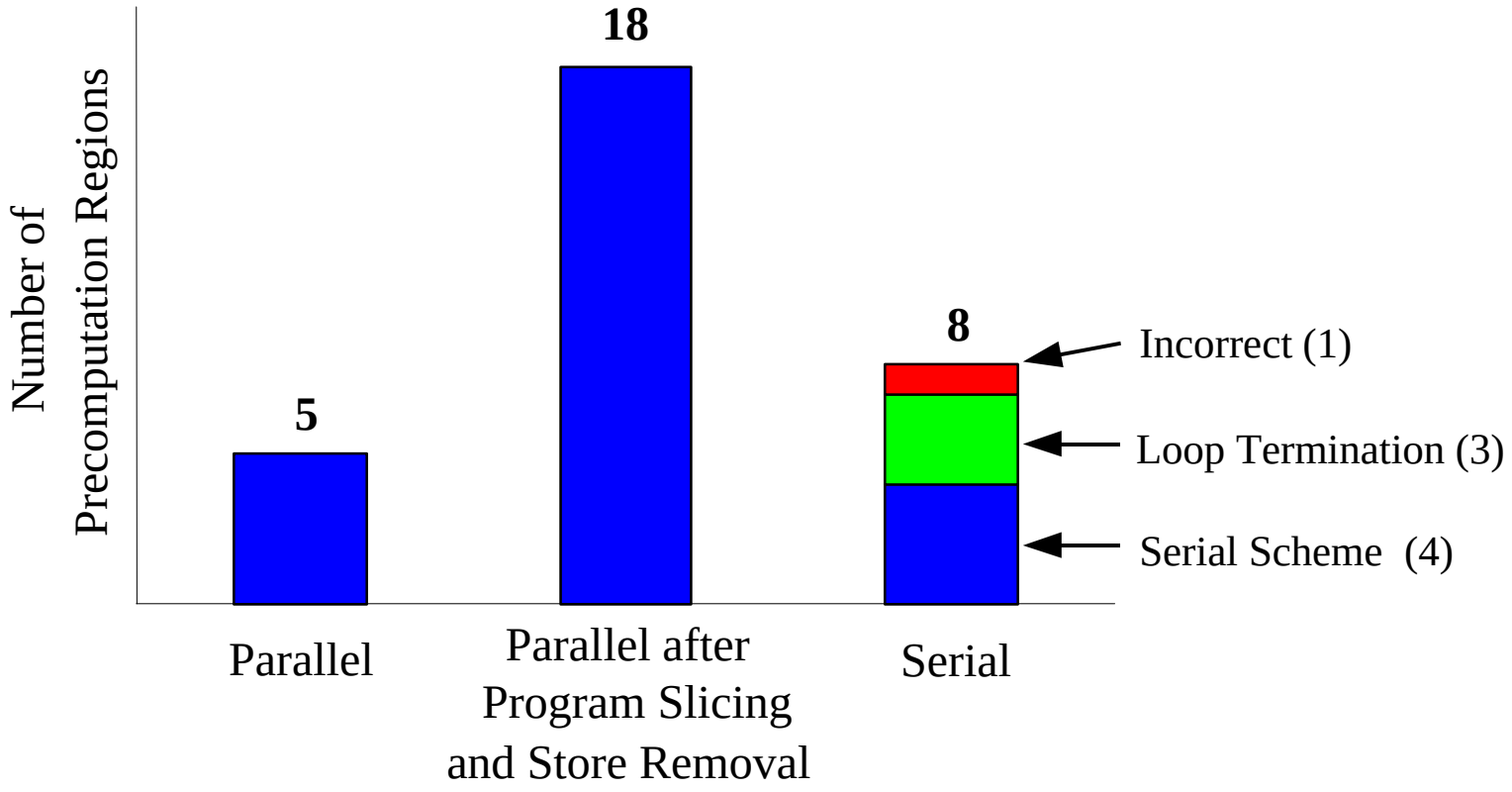


- Selecting initiation scheme makes 26%-51% difference

- Our scheme selector always picked up the best scheme

Accuracy of Precomputation Code

- 31 precomputation regions
- Store removal is wrong for 3 loops
- Speculative loop parallelization is wrong for 1 loop



Outline

- Compiler-Based Speculative Precomputation
  - Optimizations
  - Implementation
  - Evaluation
- Alternate Compiler Algorithms
  - Simplify Program Slicing
  - Eliminate Profiles
- Software-Controlled Pre-Execution

Program Slicing vs. Dead Code Elimination

- Program slicing is redundant
  - Removes dead code, keeps pinned code
  - Code removed by C compiler anyways
- Eliminate program slicer
  - Apply store removal and code pinning
  - Dead code elimination removes unpinned code
- But prefetch conversion requires slicing analysis ...
  - Perform basic slicing analysis only
  - No need for advanced analysis

Compiling without Profiles

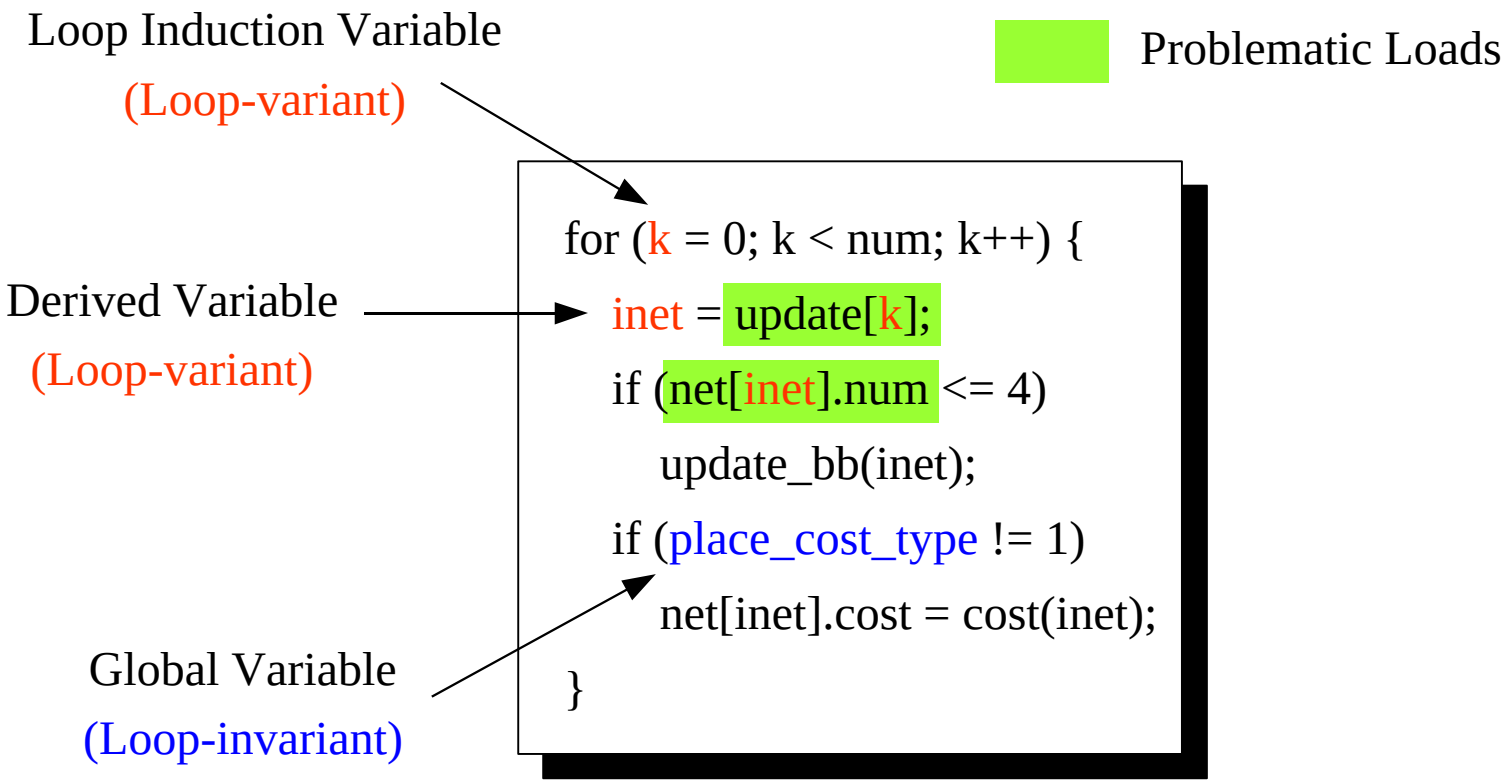
- Profiles are cumbersome
  - Require profile runs
  - Sensitive to program inputs
- Replace with compile-time heuristics
- Problematic load identification

  ➡ Assume all loop-varying references cache miss

- Loop work estimation

  ➡ Assume all inner-most loops are small

Problematic Load Identification

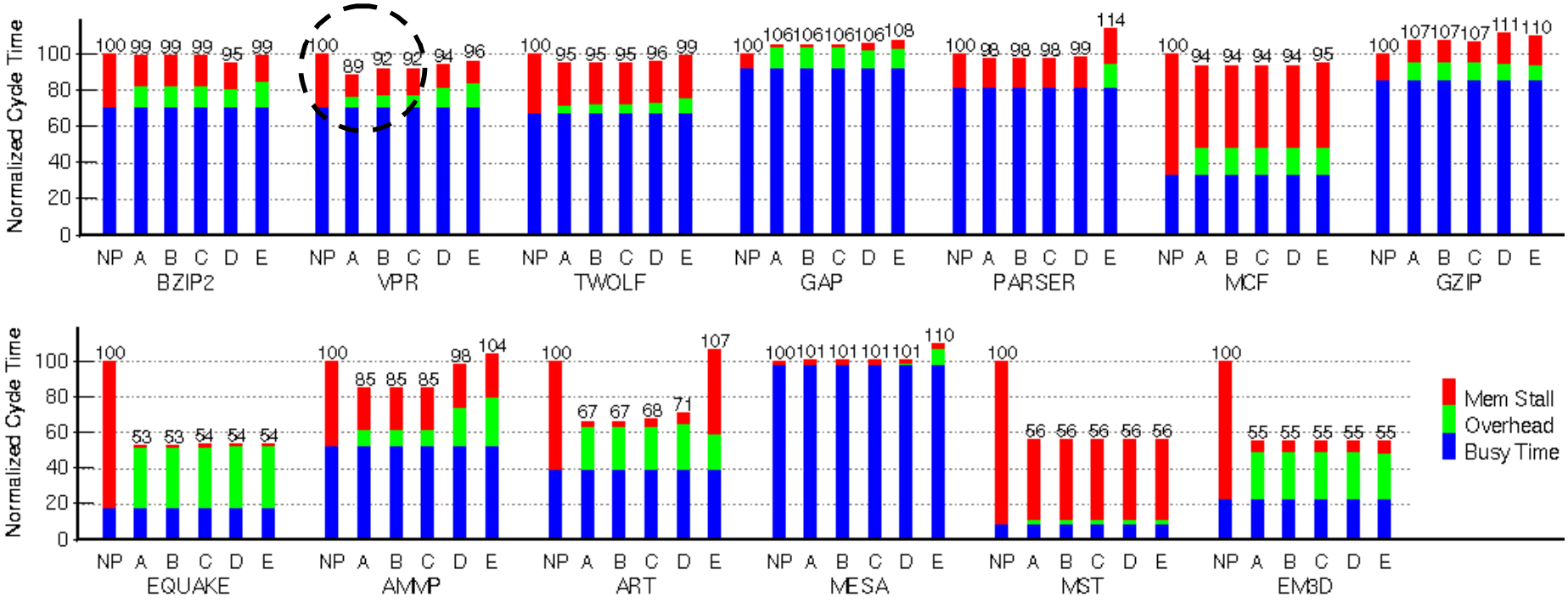


Alternate Compilers

- 5 incremental compilers:

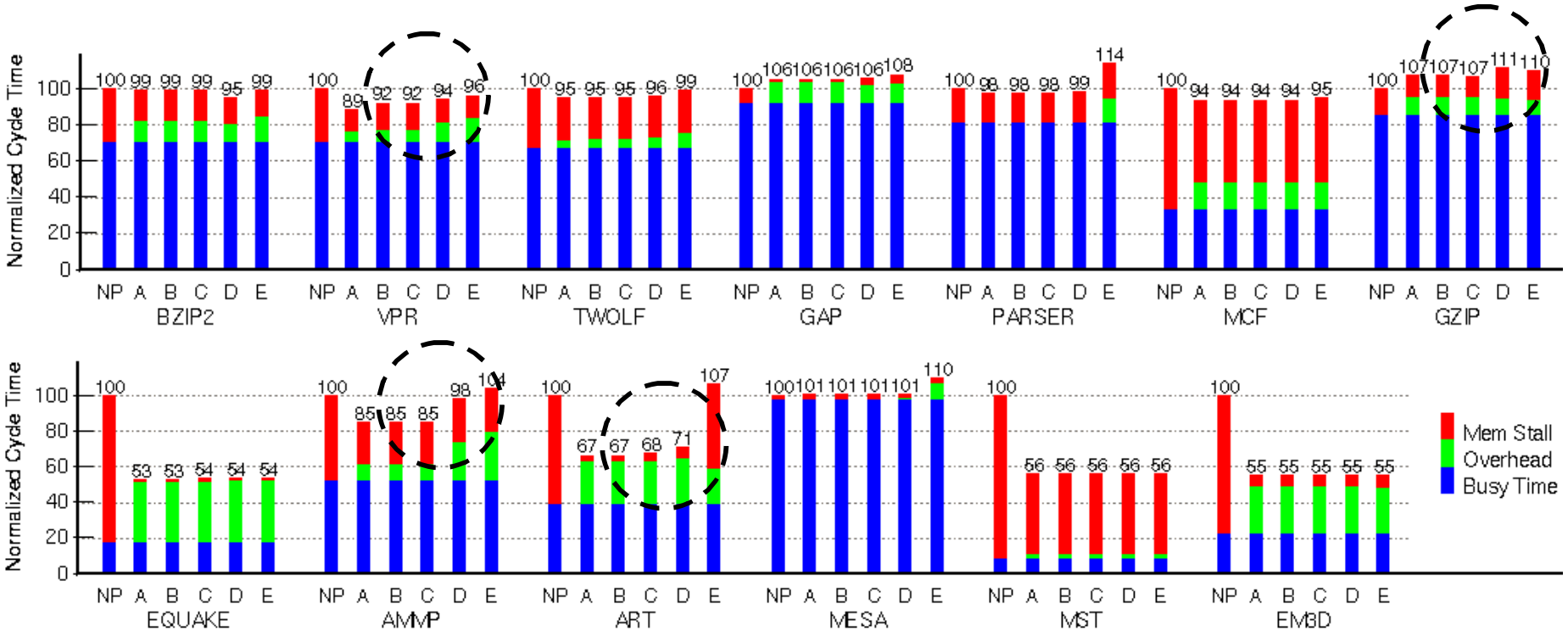
	Code Removal	Prefetch Converter	Problematic Load Identifier	Loop Work Estimator
A	Unravel	Unravel	SimpleScalar	SimpleScalar
B	C Compiler	Unravel	SimpleScalar	SimpleScalar
C	C Compiler	SUIF	SimpleScalar	SimpleScalar
D	C Compiler	SUIF	SUIF	SimpleScalar
E	C Compiler	SUIF	SUIF	SUIF

Evaluation of Alternate Compilers



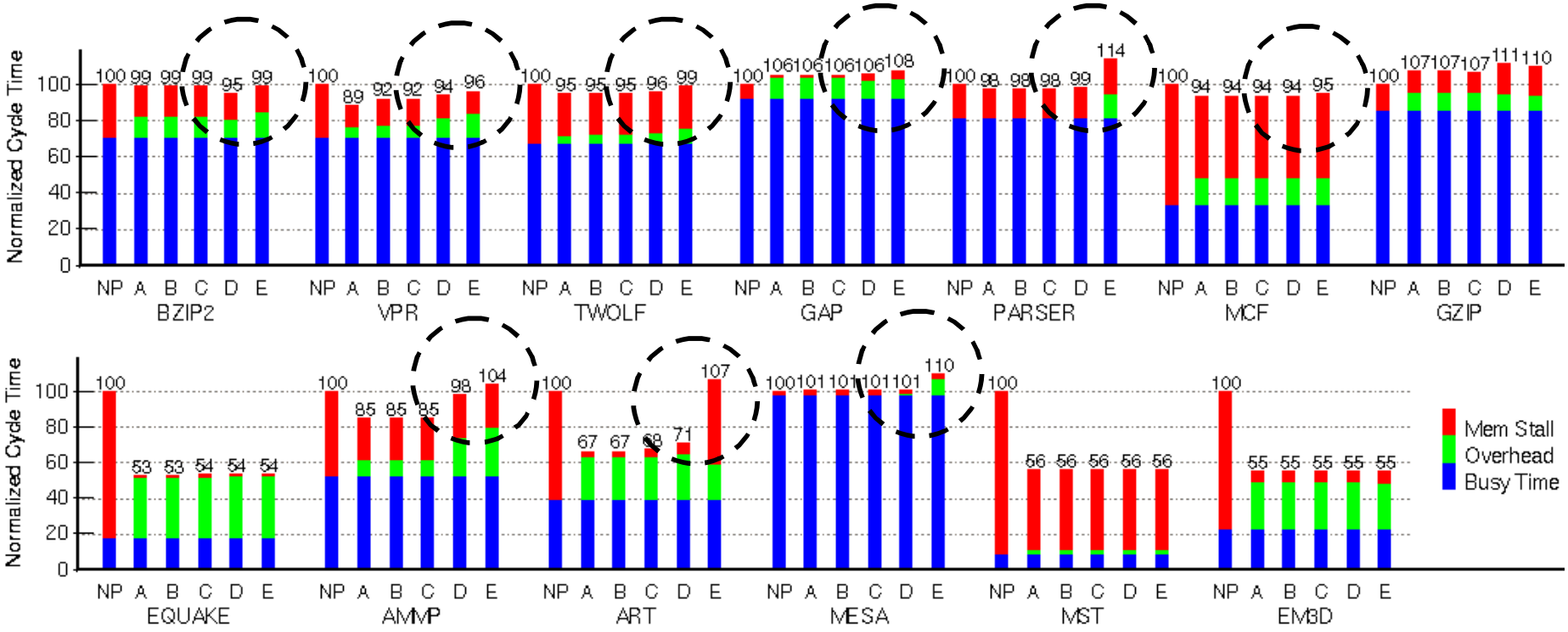
- Eliminating Unravel impacts only 1 application
- Speedup drops from 17.6% to 17.3%

Evaluation of Alternate Compilers



- Eliminating cache-miss profiles impacts 4 applications
- Speedup drops from 17.1% to 15.0%

Evaluation of Alternate Compilers



- Eliminating loop profiles impacts 9 applications
- Speedup drops from 15.0% to 7.7%

References

- Dongkeun Kim and Donald Yeung. **Design and Evaluation of Compiler Algorithms for Pre-Execution.** In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems.* October 2002.
- Dongkeun Kim and Donald Yeung. **A Study of Source-Level Compiler Algorithms for Automatic Construction of Pre-Execution Code.** To appear in *ACM Transactions on Computer Systems.* Draft available at http://www.ece.umd.edu/~yeung.

- **The SSMT Simulator** (based on SimpleScalar)**.** http://maggini.eng.umd.edu/vortex/ssmt.html

Outline

- Compiler-Based Speculative Precomputation
  - Optimizations
  - Implementation
  - Evaluation
- Alternate Compiler Algorithms
  - Simplify Program Slicing
  - Eliminate Profiles
- Software-Controlled Pre-Execution

Software-Controlled Pre-Execution (Luk 2001)

- Extract precomputation code at source level
- No code cloning
  - Main and precomputation threads execute same code
  - Smaller code size, I-cache foot print
  - Code transformations affect main thread

    (no program slicing, prefetch conversion)
- By-hand instrumentation

Sharing the Same Code

- PreExecute_Start() initiates precomputation; ignored by precomputation threads
- PreExecute_Stop() terminates precomputation; ignored by main thread

```
Void BlueRule(Vtx inserted, Vtx vslit) {
   Vertex tmp;
   Hash hash;
   for (tmp=vlist->next; tmp; tmp=tmp->next) {
      PreExecute_Start(END_FOR);
      ...
      hash = tmp->edgehash;
      dist = (int)HashLookup(inserted, hash);
      ...
      PreExecute_Stop();
END_FOR:
   }
   PreExecute_Stop();
}
```

Initiating Multiple Precomputation Threads

- Expose multiple initiation points by loop unrolling

```
Void BlueRule(Vtx inserted, Vtx vslit) {
   Vertex tmp;
   for (tmp=vlist->next; tmp; tmp=tmp->next) {
      next_1 = tmp->next;
      PreExecute_Start(UNROLL_1);
      if (next_1) {
         PreExecute_Start(UNROLL_2);
      }
      dist = (int)HashLookup(inserted, tmp->edgehash);
      PreExecute_Stop();
      ...
      next_1 = tmp->next;
UNROLL_1:
      if (next_1) {
         ...
         dist = (int)HashLookup(inserted, next_1->edgehash);
         PreExecute_Stop();
         ...
      } else
         break;
UNROLL_2:
      tmp = next_1;
   }
   PreExecute_Stop();
}
```