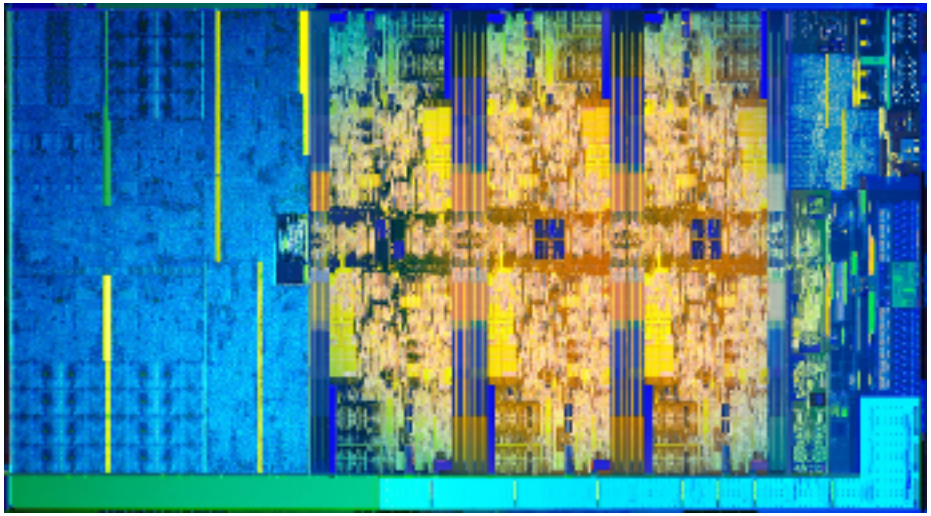
LEHIGH UNIVERSITY

Universal Support for Scoped Memory Access Instrumentation

Chris Kjellqvist, Pantea Zardoshti, Michael Spear

- Advances in software and hardware research brings new memory features
  - Transactional Memory
  - Non-Volatile Memory (NVM)
  - Non Uniform Memory Access (NUMA)
  - Guarded Memory



Intel Coffee Lake Die Layout

- A transaction is a sequence of steps executed by a single thread.
- Atomic transaction
  - Commit: memory writes take effect
  - Abort: effects rolled back

```
void f(int* n){
    if (*n == 4)
        *n = *n + 1;
}
int a = 4;
f (&a);
```

- Consider the previous function, but when two threads execute the function at the same time

*n	Thread 1	Thread 2
4		if (*n == 4)
4	if (*n == 4)	
5	*n = *n +1;	
6		*n = *n +1;

```
void f(int* n){
    if (*n == 4)
        *n = *n + 1;
}
//initialized globally as 4
f (some_shared_ptr);
```

» We can make our code work by using transactions

Thread 1	Thread 2
transaction {	
	transaction {
if (*n == 4)	
	if (*n == 4)
*n = *n + 1;	
	*n = *n + 1;
	} (transaction succeeds)
} (transaction aborts)	

```
void f(int* n){
    transaction {
        if (*n == 4)
            *n = *n + 1;
    }
}
//initialized globally as 4
f (some_shared_ptr);
```

- However using current transactional memory libraries is a much more complicated process than previously shown
- Static Separation

RSTM API

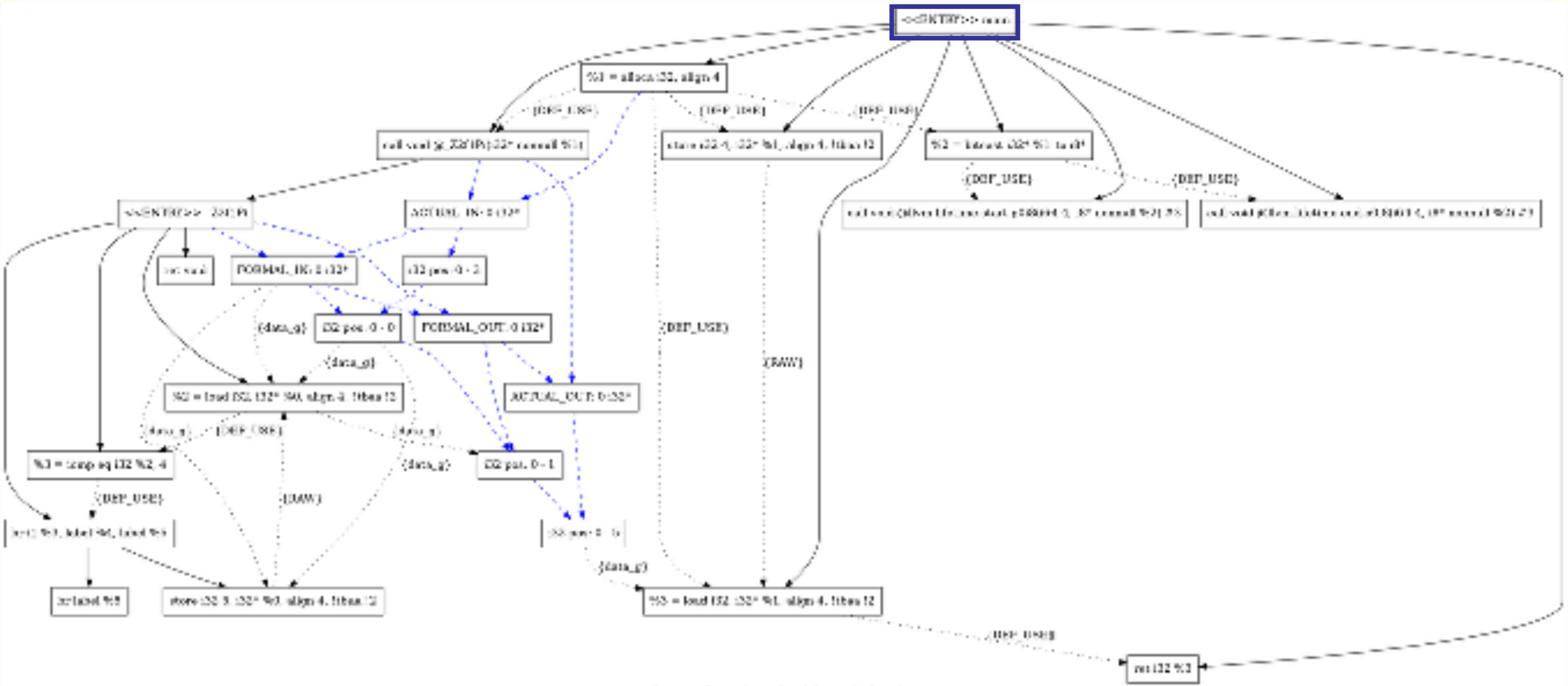
```
void f(int* n){
    TM_BEGIN(atomic) {
        int z = TM_READ(n);
        if (z == 4)
            TM_WRITE(n, z+1);
    } TM_END;
}
```

Intel TSX API

```
void f(int* n){
    int status;
    while ((status = _xbegin()) != _XBEGIN_STARTED) {
        if (*n == 4)
           *n = *n + 1;
        _xend()
     }
}
```

- To accomplish this task, we have developed a set of C++ keywords to streamline the use of TM
  - Annotate functions that you’d like to act as transactions
  - Annotate variables in a similar way

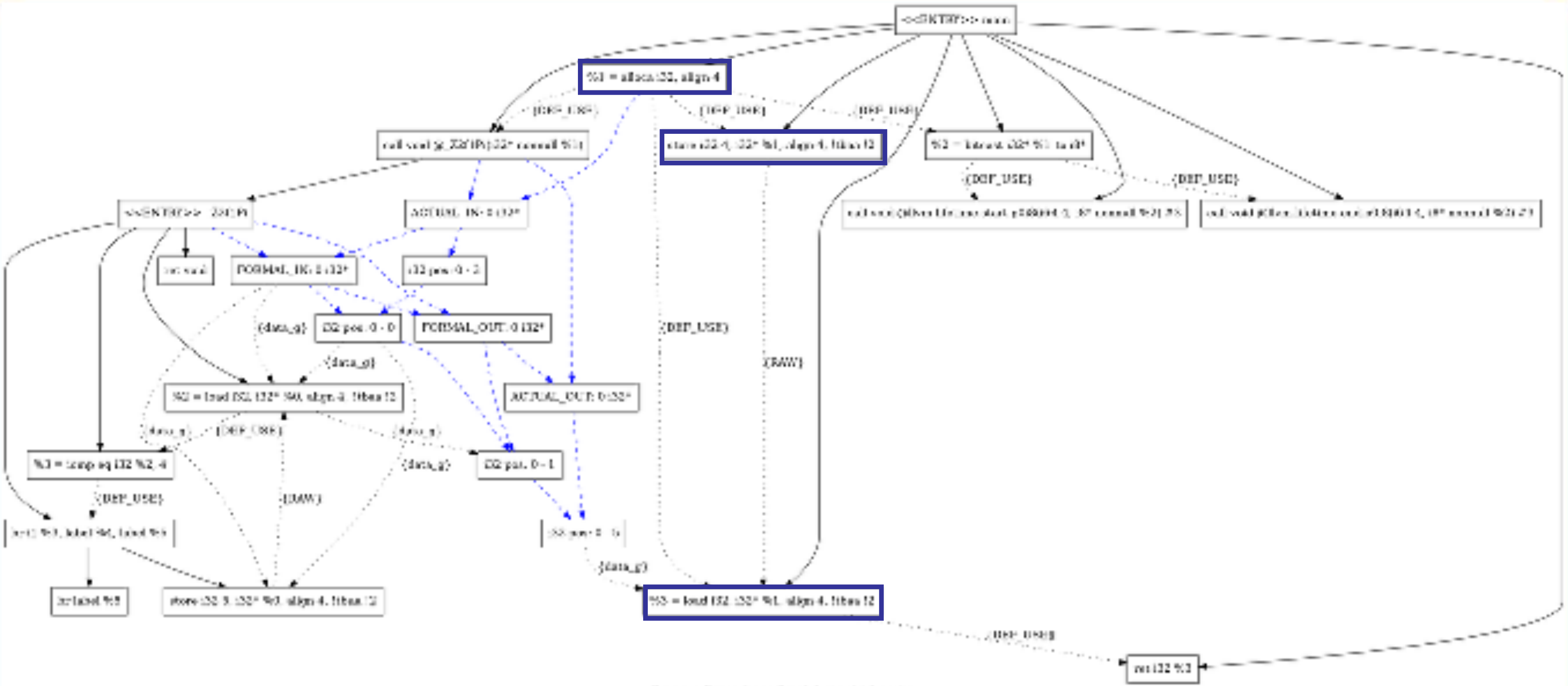
```
TX_SAFE void f1(int* n){
    if (*n == 4)
        *n = *n + 1;
}
int main() {
    TX_VAR(int) a = 4;
    TX_PTR(int) *b = malloc(4);

    f1 (&a);
    return a;
}
```


<<ENTRY>> main
%1 = alloca i32, align 4
{DEF_USE}
call void @_Z2f1Pi(i32* nonnull %1)
store i32 4, i32* %1, align 4, !tbaa !2
%2 = bitcast i32* %1 to i8*
call void @llvm.lifetime.start.p0i8(i64 4, i8* nonnull %2) #5
call void @llvm.lifetime.end.p0i8(i64 4, i8* nonnull %2) #5
<<ENTRY>> _Z2f1Pi
ACTUAL_IN: 0 i32*
ret void
FORMAL_IN: 0 i32*
i32 pos: 0 - 2
{data_g}
i32 pos: 0 - 0
FORMAL_OUT: 0 i32*
{RAW}
%2 = load i32, i32* %0, align 4, !tbaa !2
ACTUAL_OUT: 0 i32*
%3 = icmp eq i32 %2, 4
i32 pos: 0 - 1
br i1 %3, label %4, label %5
i32 pos: 0 - 1
br label %5
store i32 5, i32* %0, align 4, !tbaa !2
%3 = load i32, i32* %1, align 4, !tbaa !2
ret i32 %3

Program Dependency Graph for 'main' function.

```
void f1(int* n){
    if (*n == 4)
        *n = *n + 1;
}
```

```
int main() {
    int a = 4;
    f1 (&a);
    return a;
}
```

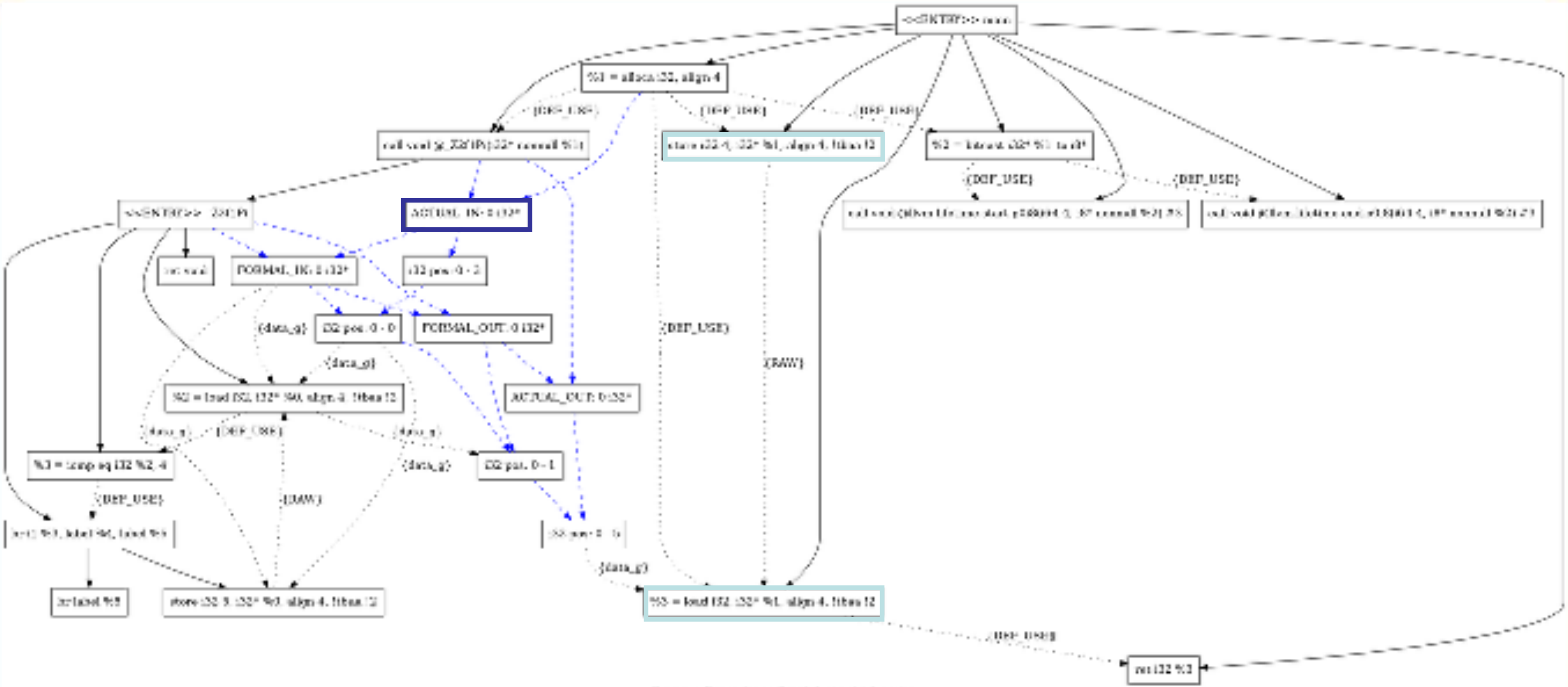

<<ENTRY>> main
%1 = alloca i32, align 4
{DEF_USE}
call void @_Z2f1Pi(i32* nonnull %1)
store i32 4, i32* %1, align 4, !tbaa !2
%2 = bitcast i32* %1 to i8*
<<ENTRY>> _Z2f1Pi
ACTUAL_IN: 0 i32*
ret void
FORMAL_IN: 0 i32*
i32 pos: 0 - 2
{data_g}
i32 pos: 0 - 0
FORMAL_OUT: 0 i32*
%2 = load i32, i32* %0, align 4, !tbaa !2
ACTUAL_OUT: 0 i32*
{RAW}
%3 = icmp eq i32 %2, 4
i32 pos: 0 - 1
br i1 %3, label %4, label %5
br label %5
store i32 5, i32* %0, align 4, !tbaa !2
%3 = load i32, i32* %1, align 4, !tbaa !2
ret i32 %3

Program Dependency Graph for 'main' function.

```
void f1(int* n){
    if (*n == 4)
        *n = *n + 1;
}
```

```
int main() {
    int a = 4;
    f1 (&a);
    return a;
}
```

Load/Store Transformation

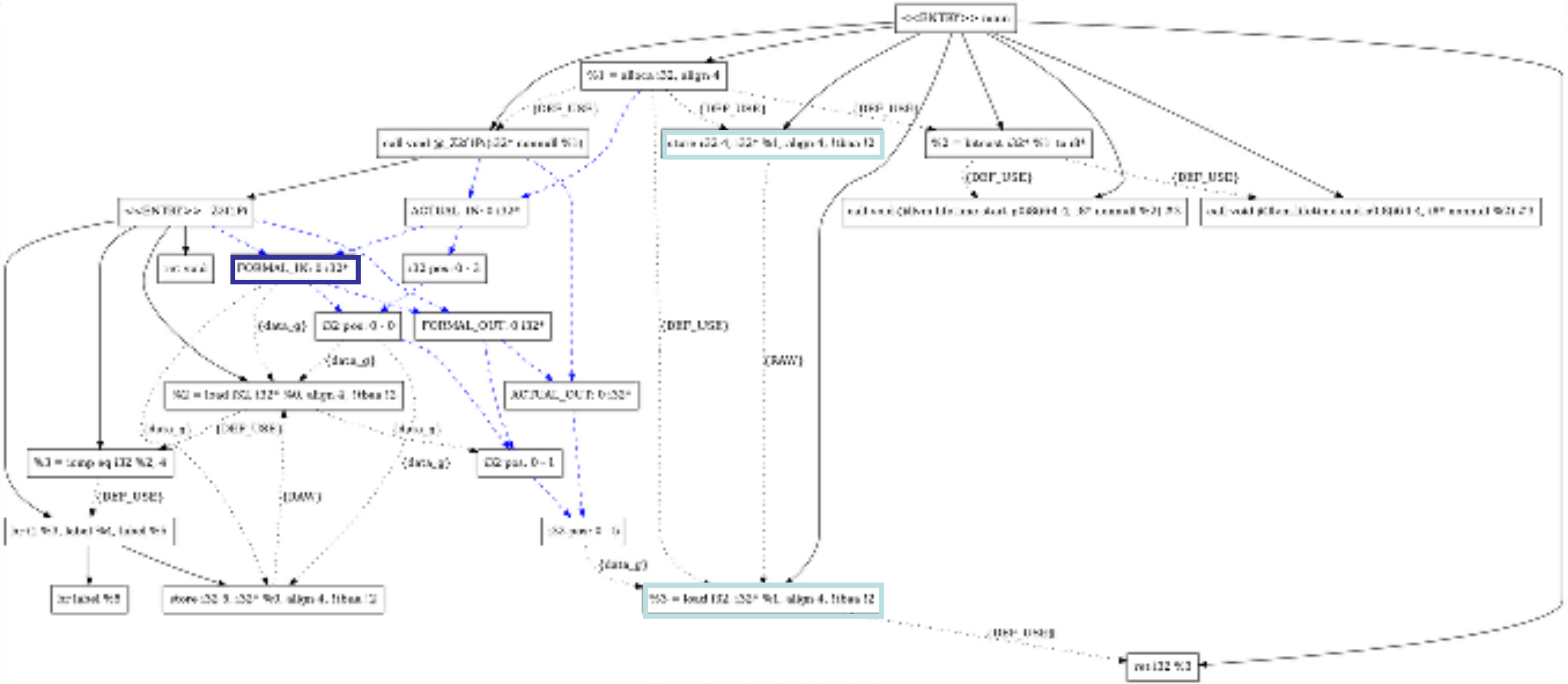


Program Dependency Graph for 'main' function.

```
void f1(int* n){
    if (*n == 4)
        *n = *n + 1;
}
```

```
int main() {
    int a = 4;
    f1 (&a);
    return a;
}
```

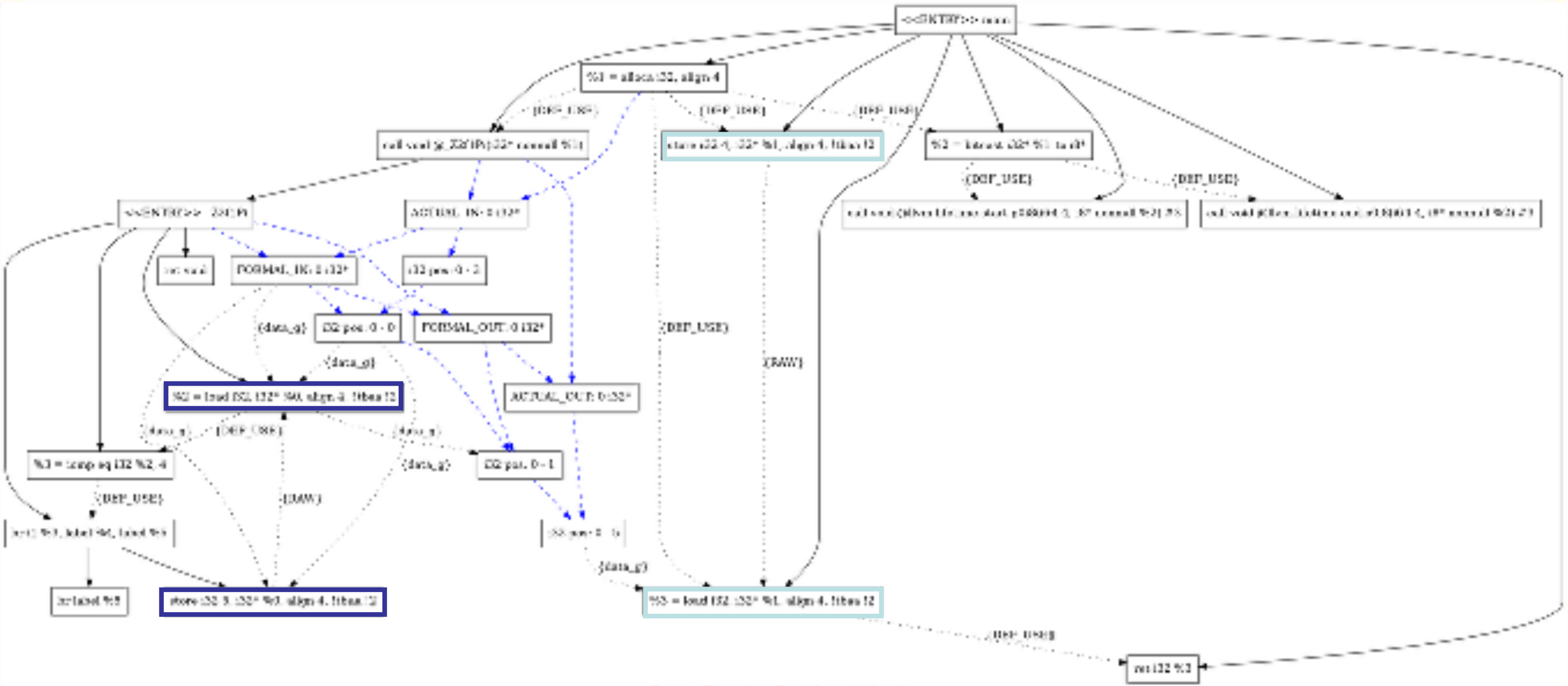
Load/Store Transformation



Program Dependency Graph for 'main' function.

```
void f1(int* n){
    if (*n == 4)
        *n = *n + 1;
}
```

```
int main() {
    int a = 4;
    f1 (&a);
    return a;
}
```

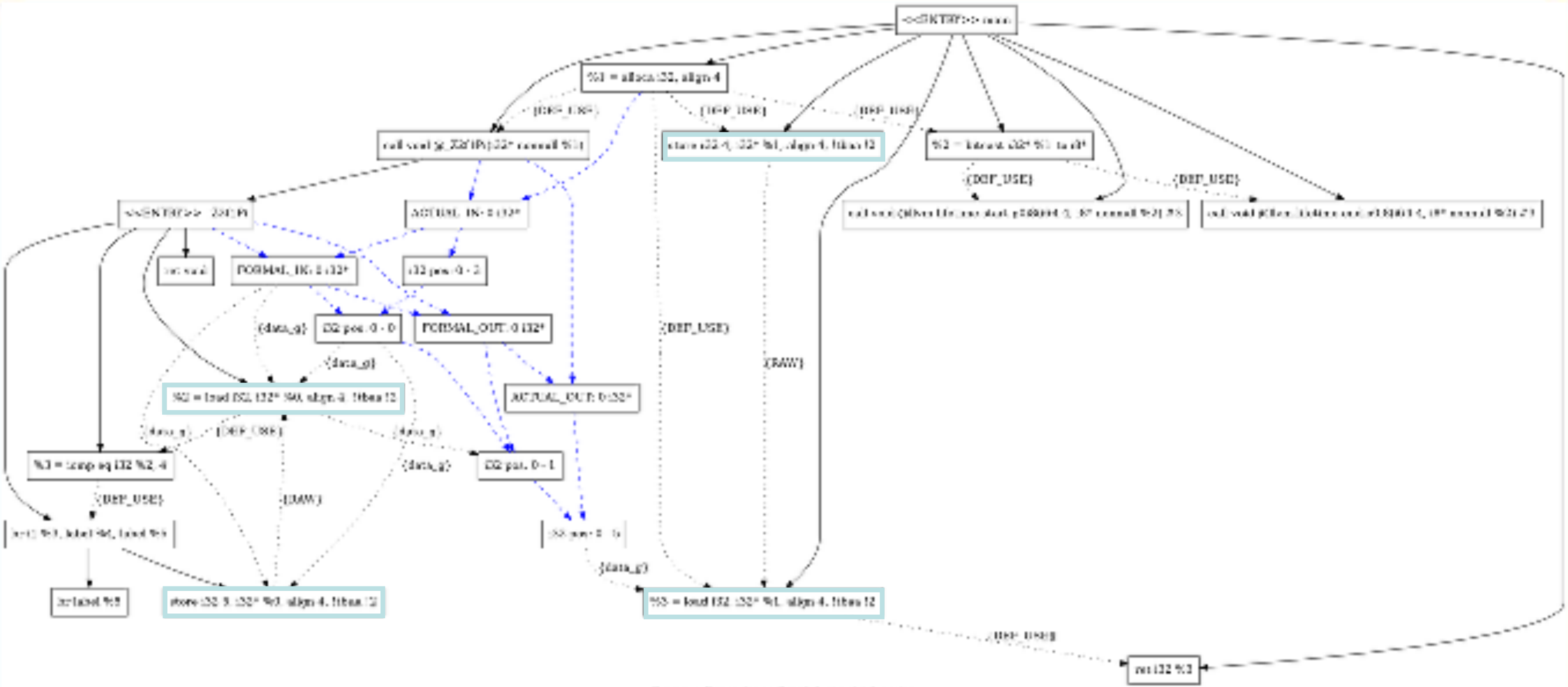


Program Dependency Graph for 'main' function.

```
void f1(int* n){
    if (*n == 4)
        *n = *n + 1;
}
```

```
int main() {
    int a = 4;
    f1 (&a);
    return a;
}
```

Load/Store Transformation

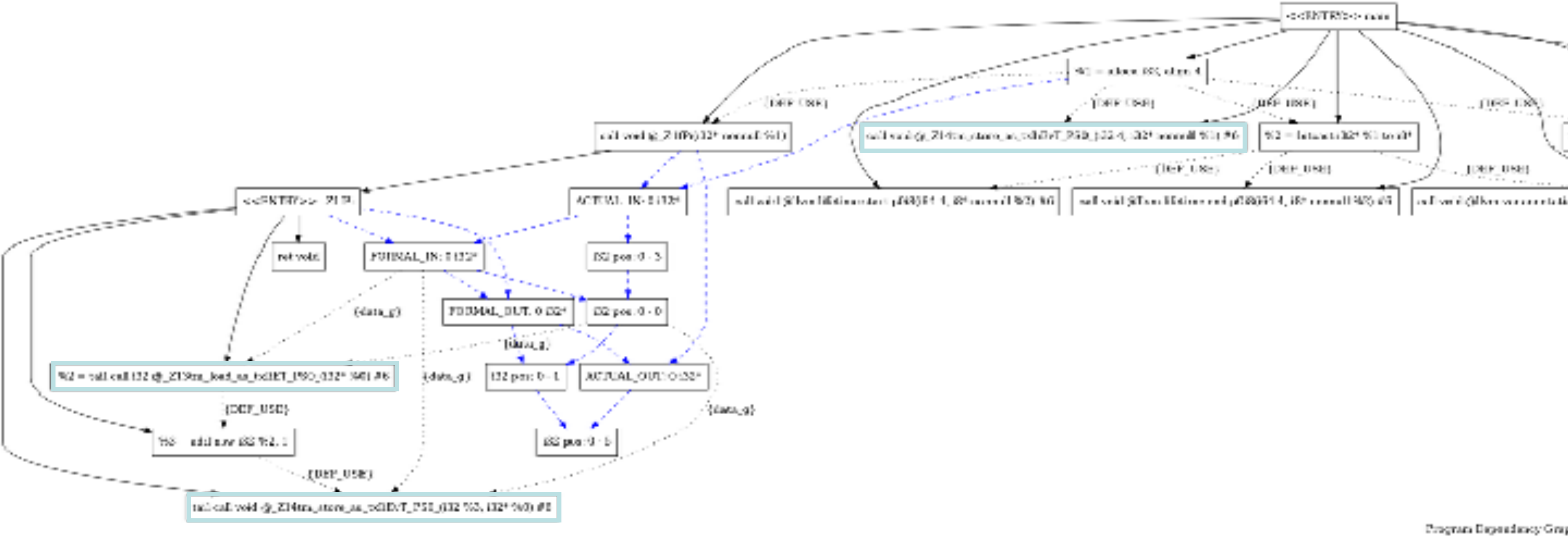


Program Dependency Graph for 'main' function.

```
void f1(int* n){
    if (*n == 4)
        *n = *n + 1;
}
```

```
int main() {
    int a = 4;
    f1 (&a);
    return a;
}
```

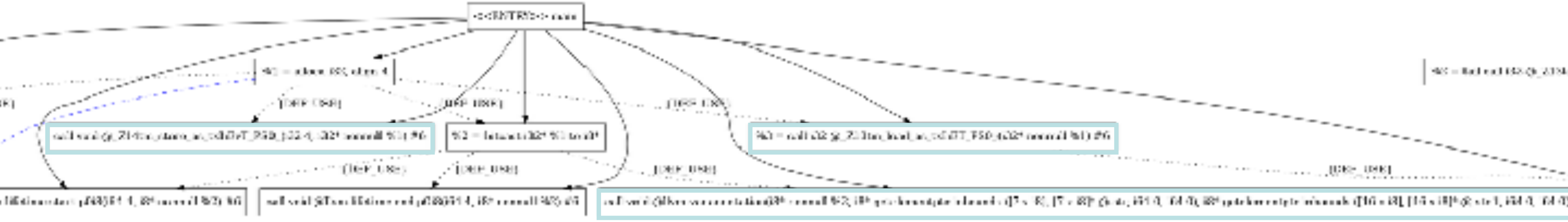
Load/Store Transformation


<<ENTRY>> main
%1 = alloca i32, align 4
{DEF_USE}
call void @_Z2f1Pi(i32* nonnull %1)
call void @_Z14tm_store_as_txDEF_PSO_(i32 4, i32* nonnull %1) #6
%2 = load i32, i32* %1, align 4
call void @llvm.lifetime.start.p0i8(i64 4, i8* nonnull %3) #6
call void @llvm.lifetime.end.p0i8(i64 4, i8* nonnull %3) #6
<<ENTRY>> _Z2f1Pi
ACTUAL_IN: 0 i32*
ret void
FORMAL_IN: 0 i32*
i32 pos: 0 - 3
FORMAL_OUT: 0 i32*
i32 pos: 0 - 0
{data_g}
%2 = tail call i32 @_Z19tm_load_as_txDEF_PSO_(i32* %0) #6
i32 pos: 0 - 1
ACTUAL_OUT: 0 i32*
%3 = add nsw i32 %2, 1
i32 pos: 0 - 5
tail call void @_Z14tm_store_as_txDEF_PSO_(i32 %3, i32* %0) #8
Program Dependency Grap

```
void f1(int* n){
    if (*n == 4)
        *n = *n + 1;
}
```

```
int main() {
    TX_VAR(int) a = 4;
    f1 (&a);
    return a;
}
```

Load/Store Transformation



Program Dependency Graph for 'main' function

```
void f1(int* n){
    if (*n == 4)
        *n = *n + 1;
}
```

```
int main() {
    TX_VAR(int) a = 4;
    f1 (&a);
    return a;
}
```

Contributions

- Ease of use to usage of TM libraries
- Optimization by data region instead of code region

Before

RSTM API

```
void f(int* n){
    TM_BEGIN(atomic) {
        int z = TM_READ(n);
        if (z == 4)
            TM_WRITE(n, z+1);
    } TM_END;
}
```

Intel TSX API

```
void f(int* n){
    int status;
    while ((status = _xbegin()) != _XBEGIN_STARTED) {
        if (*n == 4)
            *n = *n + 1;
        _xend()
    }
}
```

After

```
TX_SAFE void f1(int* n){
    if (*n == 4)
        *n = *n + 1;
}
```

```
int main() {
    TX_VAR(int) a = 4;
    f1 (&a);
    return a;
}
```

- » Transactions can reduce assumptions of multithreaded code
- » Heterogenous memory systems are hard to use consistently
- » Variable attributes let us transform loads/stores outside of transactions to assist with static separation

Summary